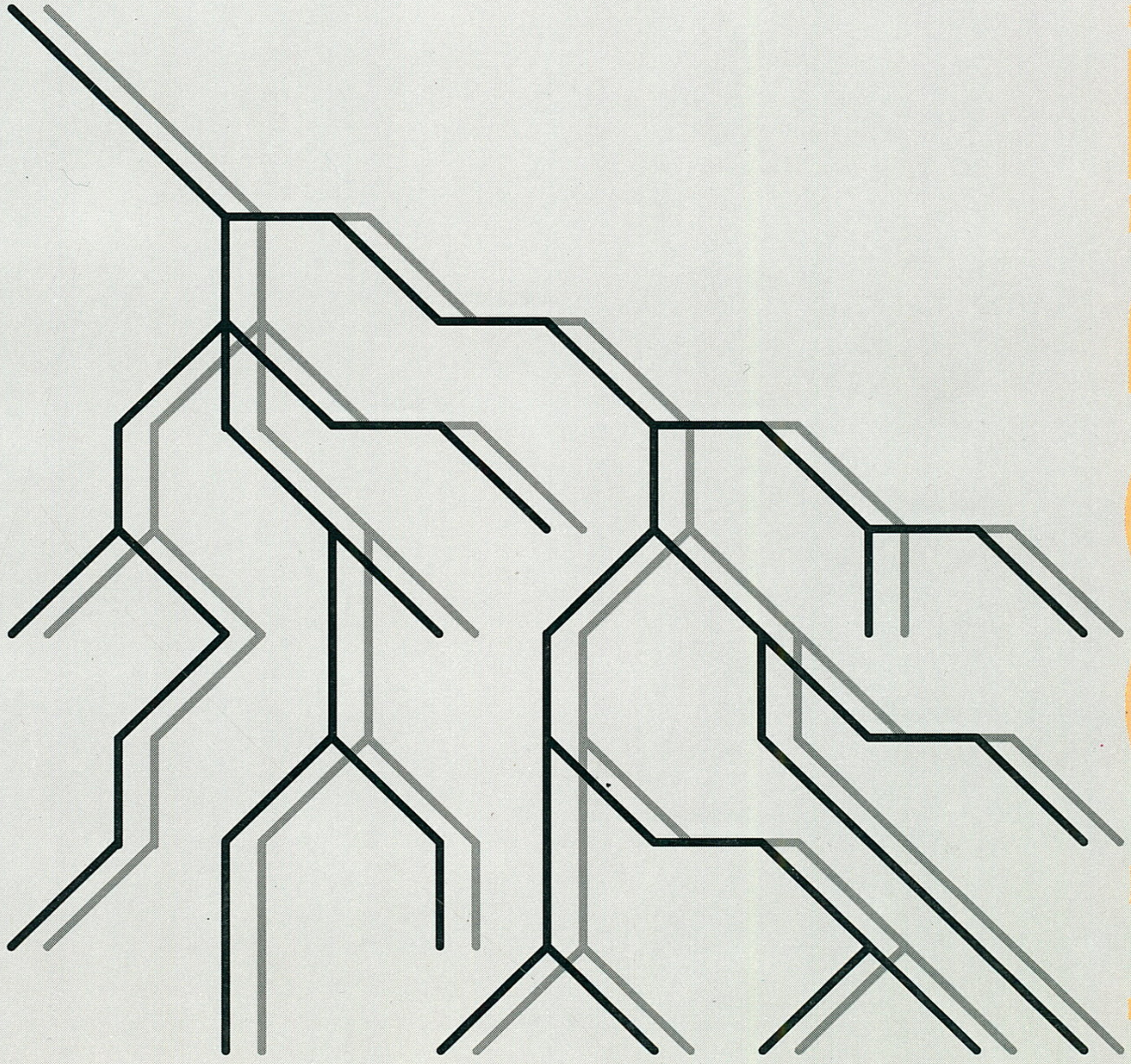
Rónyai Lajos
Ivanyos Gábor
Szabó Réka

ALGORITMUSOK



E felsőoktatási tankönyv a Művelődési és Közoktatási Minisztérium támogatásával, a Felsőoktatási Pályázatok irodája által lebonyolított felsőoktatási tankönyvtámogatási program keretében jelent meg.

ISBN 963 9132 16 0

©Rónyai Lajos, Ivanyos Gábor, Szabó Réka, 1998

Lektorálta: Elekes György

Rónyai Lajos

Ivanyos Gábor

Szabó Réka

Algoritmusok

TYPOTEX

1999

Tartalomjegyzék

Bevezetés 9

1. Algoritmikus problémák megoldása 12

1.1. A feladattól a modellig . 14

1.1.1. Közlekedési lámpák ütemezése 14

1.1.2. Arthur király civilizációs törekvései 16

1.2. Algoritmusok . 18

1.2.1. Szuperforrás keresése 18

1.2.2. Hosszú egészek párhuzamos összeadása 21

2. Rendezés 25

2.1. Keresés rendezett halmazban 27

2.2. Összehasonlítás alapú rendező módszerek 29

2.2.1. Buborék-rendezés . 30

2.2.2. Beszúrásos rendezés . 31

2.2.3. Egy alsó becslés . 33

2.2.4. Összefésüléses rendezés 35

2.2.5. A kupac adatszerkezet és a kupacos rendezés 37

2.2.6. Gyorsrendezés . 42

2.2.7. A k-adik elem kiválasztása 44

2.3. Kulcsmanipulációs rendezések 46

2.3.1. Ládarendezés (binsort) 47

2.3.2. Radix rendezés . 48

2.4. A Batcher-féle páros-páratlan összefésülés 50

2.5. Külső tárak tartalmának rendezése 51

2.5.1. Összefésüléses rendezés külső tárakon 52

3. Keresőfák 57

3.1. Bináris fák . 58

3.2. Bináris keresőfák, naiv algoritmusok 60
3.3. 2-3-fák . 64
3.4. B-fák . 69
3.5. AVL-fák . 71
3.6. További megjegyzések kiegyensúlyozott fákról 77
3.7. Egy önszervező megoldás: az S-fák 80
3.8. Szófák . 83

4. Hash-elés és szekvenciális keresés 86
4.1. A hash-elés alapjai . 87
4.1.1. Vödrös hash-elés . 88
4.1.2. Nyitott címzés . 90
4.2. Hash-függvények . 95
4.3. Hash-elés kontra keresőfák . 98
4.4. Szekvenciális keresés . 99

5. Információtömörítés 102
5.1. A Huffman-kód . 102
5.2. A Lempel–Ziv–Welch-módszer 106

6. Gráfalgoritmusok 110
6.1. Bevezetés . 110
6.1.1. Alapfogalmak, jelölések 111
6.1.2. Gráfok ábrázolásai . 113
6.2. A legrövidebb utak problémája (egy forrásból) 115
6.2.1. Dijkstra módszere . 116
6.2.2. A Bellman—Ford-módszer 120
6.3. Floyd módszere az összes csúcspár közötti távolság meghatározására 122
6.3.1. Floyd módszere . 123
6.3.2. Tranzitív lezárás . 125
6.3.3. Egy alkalmazás: centrum keresése irányított gráfban . . . 126
6.4. Mélységi bejárás . 127
6.4.1. Irányított gráfok mélységi bejárása 128
6.4.2. Irányított körmentes gráfok (DAG-ok) 135
6.4.3. Erősen összefüggő (erős) komponensek 141
6.4.4. Irányítatlan gráfok mélységi bejárása 144
6.5. A szélességi bejárás . 146
6.6. Minimális költségű feszítőfák . 151
6.6.1. Prim módszere . 156
6.6.2. Kruskal módszere . 160

6.6.3. Az UNIÓ-HOLVAN adatszerkezet 162
6.6.4. Megjegyzések . 166
6.7. Maximális párosítás páros gráfokban 167
6.7.1. A magyar módszer . 168
6.8. Maximális folyamok hálózatokban 171
6.8.1. Kapcsolat a minimális vágással: a Ford–Fulkerson-tétel . . 174
6.8.2. A Ford–Fulkerson-algoritmus 178
6.8.3. Edmonds–Karp és Dinic algoritmusai 179
6.8.4. Alkalmazások . 183

7. Turing-gépek **190**
7.1. A Turing-gép fogalma . 191
7.2. Idő- és tárigény . 197
7.3. Néhány szimuláció . 198
7.4. A kiszámíthatóság alapfogalmai 202
7.5. Az univerzális Turing-gép 205
7.6. Alapvető kiszámíthatatlansági tételek 208
7.6.1. A diagonális nyelv – egy nem rekurzíve felsorolható nyelv 208
7.6.2. Az univerzális nyelv – egy rekurzíve felsorolható, de nem rekurzív nyelv . 209
7.7. Összefüggések a kiszámíthatósági fogalmak között 210
7.7.1. Rekurzivitás és rekurzíve felsorolhatóság 211
7.7.2. Függvények és halmazok (nyelvek) 213
7.8. További eldönthetetlen problémák 215
7.8.1. A Megállási probléma (Halting problem) 216
7.8.2. Hilbert 10. problémája 217
7.8.3. A Dominóprobléma . 222
7.8.4. Post megfeleltetési problémája 226
7.8.5. Egy nyitott kérdés: a kongruens számok felismerése . . . 227
7.9. Kolmogorov-bonyolultság 229
7.10. A közvetlen elérésű gép (RAM) 239

8. Az NP nyelvosztály **246**
8.1. Idő- és tárkorlátok . 247
8.2. Tár–idő-tétel, nevezetes nyelvosztályok 250
8.3. Nemdeterminisztikus Turing-gépek; az NP nyelvosztály 255
8.4. Néhány NP-beli nyelv . 262
8.4.1. 3 színnel színezhető gráfok 262
8.4.2. Hamilton-körrel rendelkező gráfok 262
8.4.3. Síkba rajzolható gráfok 263

8.4.4. A prímszámok nyelve 265
8.4.5. A felismerés és a keresés kapcsolata (prímtényezős felbontás) . 266
8.5. Karp-redukció, NP-teljesség 268
8.6. A SAT nyelv és a Cook–Levin-tétel 272
8.7. További NP-teljes feladatok 275
8.7.1. Konjunktív normálformájú formulák kielégíthetősége és a 3-SAT . 276
8.7.2. 3 színnel színezhető gráfok 278
8.7.3. Maximális méretű független pontrendszer gráfokban . . . 280
8.7.4. A 3 dimenziós házasítás és az X3C feladat 282
8.7.5. Hamilton-kört tartalmazó gráfok és az Utazó ügynök probléma . 285
8.7.6. A Hátizsák feladat és néhány más rokon probléma 289
8.7.7. Lineáris programozás 292
8.7.8. Minimális késésszámú ütemezés 296

9. Néhány általános algoritmus-tervezési módszer 297
9.1. Elágazás és korlátozás 299
9.2. Dinamikus programozás 302
9.3. Közelítő algoritmusok 305
9.4. Véletlent használó módszerek 310
9.4.1. Az RP nyelvosztály 314
9.4.2. Prímtesztelés . 316
9.4.3. Nagy prímszám keresése 320
9.5. Prekondícionálás . 321

10. Nyilvános kulcsú titkosírások 328
10.1. Kriptográfia - a titkosírások tudománya 328
10.2. Nyilvános kulcsú kriptográfia 331
10.3. A Rivest–Shamir–Adleman- (RSA-) kód 332

Tárgymutató 336

Bevezetés

Amiről beszélni fogok, az sem bonyolult, sem perlekedő nem lesz. JOHN L. AUSTIN

Ez a könyv *algoritmusokról* szól. Az algoritmus fogalmáról – azt gondoljuk – már minden olvasónknak van valamilyen képe. Pontos meghatározásra nem szeretnénk vállalkozni, elsősorban azért nem, mert a szó a szakmán belül is több különböző értelemben használatos. E változatok közös magjának érzékeltetésére olyan szinonimákat említhetünk, mint *eljárás*, *recept*, *módszer*. Jól meghatározott lépések egymásutánja, amelyek már elég pontosan, egyértelműen megfogalmazottak ahhoz, hogy gépiesen végrehajthatók legyenek.

Az „algoritmus" szó eredetéért a IX. századba és az Arab Kalifátus csillogó fővárosába, Bagdadba kell visszatekintenünk. Itt dolgozott az Al Khvarizmi (Khorezmből való) néven ismert Mohamed ibn Músza, aki több nevezetes tudományos könyvet írt. Ezek egyike (a latin fordításban fennmaradt *De Numero Indorum*) a decimális számokkal való hindu eredetű számolási eljárásokat írja le. Lényegében azokat, amelyeket az elemi iskolában tanultunk az egészekkel való alapműveletek elvégzésére. Elsősorban világos stílusának és a pontos, részletes indoklásoknak köszönhetően Al Khvarizmi könyvét évszázadokig forgatták a középkori Európában. Az algoritmus szó a szerző nevének a latin fordítások által eltorzított változata. Sokáig ezeket az indiai eredetű számolási szabályokat értették alatta.

A számítógépes algoritmus fogalma szorosan kapcsolódik a program fogalmához. Az egyik lehetséges megközelítés, amivel később részletesebben is megismerkedünk, lényegében azonosítja a kettőt. Az algoritmusok elmélete, ahogy ma állnak a dolgok, jóval kevesebbet vállal fel, mint amennyit az előbbi ambiciózus meghatározás sugall. Ez a terület főként a kisebb, építőkő jellegű problémákkal foglalkozik. Nemigen szokás algoritmusnak nevezni egy több tízezer utasításból álló adatbáziskezelő programot vagy annak logikai vázát.

Az algoritmika konstruktív iránya elsősorban akkora feladatokkal foglalkozik, melyek megoldása legfeljebb néhány száz C-sorban kódolható. A megoldást jelentő módszerek tervezésének nélkülözhetetlen része a módszerek elemzése. Az elemzés során arra keresünk választ, hogy algoritmusaink mennyire hatékonyak.

Az összetett, igazán nagyméretű feladatokkal a *szoftvertechnológia* (a szokásos angol szakkifejezésekkel *software engineering*, illetve *software technology*) foglalkozik. A problémák méretének növekedtével egészen más kérdések kerülnek előtérbe. Ilyenek például a tervezendő rendszer áttekinthetősége, felbonthatósága emberszabású részekre, az így kapott darabok összeillesztése, tesztelése, stb. Mi itt nem foglalkozunk ezekkel a kérdésekkel.

Algoritmusokkal és adatszerkezetekkel kapcsolatos ismeretekre, készségekre szüksége van mindenkinek, aki komolyan foglalkozik programozással és programok tervezésével. Ennek megfelelően kialakult egy eléggé letisztult törzsanyag, amit világszerte oktatnak a számítástechnikai, informatikai képzést nyújtó egyetemi szakokon. Elsődleges célunk volt ennek az anyagnak a feldolgozása.

A bevezető jellegű első fejezetben egyszerű példákon keresztül igyekeztünk érzékeltetni az algoritmusok tervezésének és elemzésének folyamatát. Ezek játékos, könnyen áttekinthető példák, amelyek segítségével az olvasó megismerkedhet a terület szemléletének, stílusának alapjaival. Ennek a résznek a fő célja az anyag befogadásához szükséges beállítottság kialakítása.

A 2-5. és részben a 6. fejezetben a ma már klasszikusnak számító alapvető adatszerkezetekkel és algoritmusaikkal foglalkozunk. A központi témák itt a rendezés és a keresés. A legérdekesebb rendező algoritmusok tárgyalása után a fa-jellegű, majd pedig a hash-elésen alapuló tárolási/keresési technikákról lesz szó. Ezt követően az információtömörítés két alapvető módszerét vesszük szemügyre (5. fejezet). A hatodik fejezet gráfokkal kapcsolatos algoritmusainak egy részét is szokás az adatszerkezetek témakörébe sorolni. Ilyenek a gyors bejáró módszerek és a rövid utak keresésére szolgáló eljárások. A gráfokról szóló részben olyan problémák is helyet kaptak, amelyek a kombinatorikus optimalizálás alapjaihoz tartoznak (hálózati folyamok, párosítások).

A hetedik fejezetet a Turing gépeknek szenteltük. Ennek az alapvető gépmodellnek az ismertetése után a kiszámíthatóság elemei következnek (rekurzivitás, eldönthetetlenség). Itt foglalkozunk a Kolmogorov-bonyolultság első tulajdonságaival, és itt kapott helyet a közvetlen elérésű gép definíciója is.

A nyolcadik fejezetbe kerültek az algoritmusok bonyolultságával kapcsolatos alapvető eredmények. Az idő- és tárkorlátokkal megadott nyelvosztályok bevezetése után az NP feladatosztály tárgyalására térünk. Komoly figyelmet szentelünk az osztály nehéz feladatainak, az NP-teljes nyelveknek.

A kilencedik fejezetben algoritmustervezési módszerekkel foglalkozunk. Olyan általános technikák ezek, amelyek feladatok széles körére alkalmazhatók. Példákon keresztül mutatunk be néhány ilyen megközelítést. Ezek a mohó módszer, az elágazás és korlátozás, a dinamikus programozás és a prekondícionálás. Fontosságuknak megfelelően kiemelten foglalkozunk a véletlent használó mód-

szerekkel.

Az utolsó fejezetben rövid ízelítőt adunk a kriptográfiából, az illetéktelen hozzáféréssel szemben biztonságos kommunikáció elméletéből. Ismertetjük a gyakorlatban is népszerű RSA kriptográfiai rendszert; ez lehetőséget ad több korábban tárgyalt fogalom, ismeret alkalmazására.

Az anyag feldolgozásakor építünk az elemi függvényekkel, gráfokkal és egészek oszthatóságával kapcsolatos alapvető tényekre. Feltételezzük még, hogy az olvasó ismeri egy általános célú programozási nyelv (mint pl. a Pascal vagy a C) utasításait.

A könyvben $\mathbb{Z}$ jelöli az egész számok és $\mathbb{R}$ a valós számok összességét. A nemnegatív egész, illetve valós számok halmazára a $\mathbb{Z}^+$, illetve $\mathbb{R}^+$ jelekkel hivatkozunk. Ha $f(x_1, x_2, \ldots, x_k)$ és $g(x_1, x_2, \ldots, x_k)$ az $(\mathbb{R}^+)^k$ egy részhalmazán értelmezett valós értékeket felvevő függvények, akkor $f = O(g)$ jelöli azt a tényt, hogy vannak olyan $c, n > 0$ állandók, hogy $|f(x_1, x_2, \ldots, x_k)| \leq c|g(x_1, x_2, \ldots, x_k)|$ teljesül, ha $x_i \geq n$ minden $i = 1, 2, \ldots, k$ esetén.

Legyenek $f(n)$ és $g(n)$ a pozitív egészeken értelmezett valós értékű függvények. Ekkor az $f = o(g)$ jelöléssel rövidítjük azt, hogy $f(n)/g(n) \to 0$, ha $n \to \infty$.

Gyakran fogunk *tömbökkel* dolgozni, mégpedig legtöbbször természetes számokkal indexelt tömbökkel. Például $A[i : j]$ jelenti az A elnevezésű tömbnek az i indexűtől a j indexű eleméig terjedő részét; $A[i]$ pedig a tömb i-indexű elemére utal.

Végezetül köszönetet mondunk mindazoknak, akik bátorításukkal, a kézirathoz fűzött megjegyzéseikkel, tanácsaikkal támogatták munkánkat. Különösen sokat köszönhetünk Babai Lászlónak, Bródy Ferencnek, Demetrovics Jánosnak, Dömötör Adrienne-nek, Elekes Györgynek, Friedl Katalinnak, Herczog Sándornénak, Kovács Annamáriának, Szabó Lászlónak, Vank Ágnesnek és Vámos Tibornak.

Budapest, 1998. július 20.

A szerzők

1.

Algoritmikus problémák megoldása

A tudományok nem próbálnak megmagyarázni semmit, alig-alig próbálkoznak értelmezéssel; főleg modelleket készítenek... Egy ilyen matematikai konstrukciónak csak és kizárólag az adja a létjogosultságát, hogy használható. NEUMANN JÁNOS

Az egyik legfontosabb célunk, hogy a hatékony algoritmusok tervezésébe bevezessük az olvasót. Mint a problémamegoldás általában, ez is sokszínű, többféle képességet igénylő kreatív folyamat. A problémával kapcsolatos ismeretek mellett komoly szerep jut az ötleteknek. Ezek olykor egyszerűek, mintegy maguktól adódnak, máskor komolyan meg kell dolgoznunk értük. Néha a megoldás szinte kiolvasható magából a feladatból, de az is előfordul, hogy csak komoly szellemi erőfeszítés árán jutunk előbbre. Nem várhatunk ezért biztos recepteket, amelyekkel minden esetben célhoz érünk. Nem vagyunk ugyanakkor teljesen támasz híján sem. Mára az algoritmika a számítógépes tudományok eszközökben és módszerekben gazdag területévé nőtte ki magát. Kialakultak olyan fogalmak, megközelítési módok, melyek iránytűként segíthetnek bennünket a megoldások felé vezető úton. Ezen a területen is fontos szerepet játszanak a *példák*, az értelmes és sikeres megoldások tanulságai, amelyek sokszor túlmutatnak felmerülésük esetleges keretein.

Ebben a rövid, bevezető jellegű fejezetben mi is példák segítségével szeretnénk első képet adni tárgyunkról, az algoritmusok világáról. Az algoritmikus problémák megoldását hasznos kétlépcsős folyamatként elképzelni, amit a következő egyszerű rajz szemléltet:

$$\boxed{\text{Algoritmikus probléma}} \xrightarrow{\mathcal{A}} \boxed{\text{modell}} \xrightarrow{\mathcal{B}} \boxed{\text{program}}$$

Az $\mathcal{A}$ leképezés azt a gyakran járt utat jelenti, hogy a problémát áttesszük, átfogalmazzuk egy kellően absztrakt, többé-kevésbé formalizált *modellbe*. Ez a lépés általában pontosítást és egyszerűsítést jelent. Megszabadulunk a probléma megfogalmazásában levő olyan elemektől, amelyek a megoldás szempontjából lényegtelenek. Legtöbbször arról van szó, hogy a feladatot kihámozzuk abból a tarkabarka, természetes nyelvből szőtt köntösből, amiben elénk került.

Az $\mathcal{A}$ lépés eredményének keretéül szolgáló modell sokféle lehet. Elég tágnak kell lennie ahhoz, hogy a probléma megoldásához szükséges tényezőket hűen kifejezze. Másfelől elég egyszerűnek is kell lennie azért, hogy kezelni tudjuk. A modell – úgy képzeljük – valahol a félúton van a probléma és a megoldását adó program között. A modell formalizáltsága, pontossága közbülső lépcsőt jelent a programsorok szigorú egyértelműsége felé. Az igazán jó modellek fogalmai könnyen és standard módon képezhetők le számítógépes adatszerkezetekre. Olykor már magában a modellben is helyet kapnak a megcélzott gépi környezet jellemzői. A másik oldalról nézve ezt a közbülső helyzetet azt is elvárjuk, hogy a modell mentes legyen a programnyelvi utasítások földhözragadtságától. Hogy tényleg lehetővé tegye azt, hogy a problémát valamiféle értelmes általánosság szintjén szemlélhessük. Divatos fordulattal élve a modellt tekinthetjük sokadik (legalább negyedik) generációs nyelvi eszköznek.

A $\mathcal{B}$ leképezés jelenti a *hatékony algoritmusok tervezését*, kidolgozását. Ez a folyamat többnyire a modell által adott keretekből indul. Ezen a szinten már egy *pontos algoritmikus problémával* van dolgunk. Ennek összetevői a *bemenő adatok* (más szóval *bemenet* vagy *input*) megadása; valamint az *eredménnyel*, az *outputtal* kapcsolatos követelmények.

A megoldó módszer első változatát a modell fogalmaival fejezzük ki. Ezt a nagyvonalú megoldást lehet azután már többé-kevésbé mechanizálható lépésekkel tényleges programmá finomítani. A későbbiek során legtöbbször ezzel a $\mathcal{B}$ lépéssel, annak is az első részével, a nagyvonalú megoldás kialakításával foglalkozunk. A munkának ebben a fázisában érdemes foglalkozni a kapott algoritmus *elemzésével, értékelésével*, megvizsgálva, hogy a módszer mennyire hatékony, mennyit lehetne még javítani rajta.

Az itt vázolt képben sok egyszerűsítés van. Ezek közül talán a legsúlyosabb, hogy a problémamegoldás folyamatát magabiztos, mindig csak előre való masírozásként tünteti fel. Ez a valóságban legtöbbször nem így van. Gyakran kényszerülünk arra, hogy visszaforduljunk egy zsákutcának bizonyuló irányból, és egy korábbi pontból új utat keressünk.

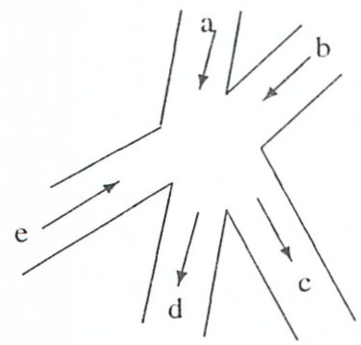
Ennyi általánosság után néhány egyszerű példával szeretnénk érzékeltetni az eddig elmondottakat. A példák bevezető jellegűek; a felmerülő fogalmakat később részletesebben fogjuk tárgyalni.

1.1. A feladattól a modellig

Itt az $\mathcal{A}$ leképezésre szeretnénk néhány példát mutatni. A modell, amibe átírjuk a problémákat, tulajdonképpen egyetlen egyszerű fogalomra, a *gráfra* épül. Ennek az egyszerű fogalomnak az alkalmazásával az eredeti problémák tiszta, pontos megfogalmazását kapjuk.

1.1.1. Közlekedési lámpák ütemezése

Képzeljük el, hogy egy útkereszteződés közlekedési lámpáinak a működését kell megterveznünk. Tegyük fel, hogy a kereszteződésben (északról indulva, órajárás szerint) az a, b, c, d, e egysávos utak találkoznak. Az a, b és e utak a kereszteződés felé, a többi pedig a kereszteződéssel ellentétes irányban egyirányú.



Tegyük fel, hogy minden értelmes, a nyilakat követő áthaladási irányhoz van egy közlekedési lámpánk. Ezek az irányok esetünkben ac, ad, bc, bd, ec és ed. Összesen tehát hat lámpánk van. Például az ac irány lámpája csak az ebben az irányban való átjutást szabályozza (tiltja, illetve engedélyezi). Ennyi a probléma leírása.

Ezután elkezdhetünk gondolkodni. Mi is a feladat tulajdonképpen? Némi tűnődés után arra jutunk, hogy a lámpákból álló rendszer számunkra érdekes állapotait egy-egy *lámpák* $\rightarrow$ {piros, zöld } függvényként foghatjuk fel. Ezzel máris egyszerűsítettük a képet, felismerve, hogy a sárga színnek nincs komoly szerepe a feladat szempontjából. Például a $jujj$ nevű állapot (függvény) lehet a következő:

$$jujj(ac) = jujj(ad) = jujj(bd) = \text{zöld},$$
$$jujj(bc) = jujj(ed) = jujj(ec) = \text{piros}.$$

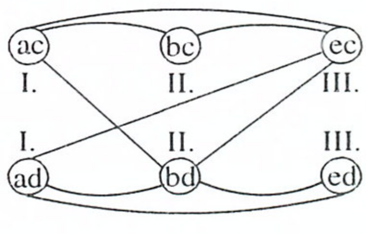
Ezt az állapotot nem szabad megengednünk, hiszen például az ac és bd irányok találkoznak. E két iránynál nem lehet egyszerre mindkét lámpa zöld. Kis további gondolkodás után oda jutunk, hogy az ilyen konfliktuslehetőségek megfoghatók

mint irányokból álló *párok*. Például (ac, bd) egy tiltott pár, (ac, ad) pedig nem. A kapcsolat szimmetrikus, nincs különbség mondjuk az (ac, bd) és a (bd, ac) párok megítélése között.

Mindezek azt sugallják, hogy próbáljuk meg gráffal ábrázolni a helyzetet. A G gráfunk csúcsai az áthaladási irányok: ac, ad, bc, bd, ec és ed. Két csúcsot akkor kössön össze él, ha az irányok konfliktusban vannak. Más szóval, ha a megfelelő két lámpa nem lehet egyszerre zöld. Egy állapot akkor engedhető meg, ha a benne levő zöld csúcsok között nem fut él. További egyszerűsítést jelent az a felismerés, hogy egy állapotot jellemezhetünk pusztán a benne levő zöld csúcsok halmazával.

Ezután megpróbálkozhatunk a feladat pontos megfogalmazásával. Az első változat így hangzik: adjunk meg minél kevesebb állapotot úgy, hogy egyfelől a megadott állapotok mind biztonságosak legyenek; másfelől ne legyen olyan áthaladási irány, amelyhez az összes megadott állapotban a piros szín tartozik. Az utóbbi feltétel azt hivatott szavatolni, hogy nem okozunk örök dugót.

A G gráf segítségével ez még egyszerűbben kifejezhető. Úgy kell a G csúcshalmazát minél kevesebb osztályba (részhalmazba) sorolni, hogy az egy osztályban levő csúcsok között ne menjen él. Esetünkben ilyen osztályozás az $\{ac, ad\}$, $\{bc, bd\}$, $\{ec, ed\}$. A rajzon római számmal tüntettük fel a csúcsok osztályának sorszámát.



Feladat: Mutassuk meg, hogy két osztállyal nem oldható meg a probléma.

Mi történt itt valójában? Az eredeti feladatot átfogalmaztuk a *gráfok* nyelvére, eltüntetve belőle olyan, a vizsgált kérdés szempontjából érdektelen elemeket, mint „közlekedés", „út", stb. Ebben a modellben lényegében csak csúcsok és őket összekötő élek szerepelnek. Valamivel pontosabban egy $G = (V, E)$ gráf két összetevőből áll. Az egyik a csúcsok (pontok) V halmaza, a másik pedig E, az élek összessége. Az E halmaz bizonyos V-beli párokból áll. Rajzon szemléltetve a csúcsokat pontokkal ábrázoljuk, az éleknek megfelelő párok tagjait pedig vonallal összekötjük. Az átfogalmazás után az eredeti feladat egy klasszikus gráfelméleti probléma algoritmikus változatához vezet.

Definíció (színezés, kromatikus szám):
A G gráf k-színezésén egy $f : V \to \{1, \ldots, k\}$ *leképezést értünk, melyre ha* $(v, w) \in E$, *akkor* $f(v) \neq f(w)$. *A G kromatikus száma a legkisebb k érték, melyre van G-nek k-színezése. A G kromatikus számának a jele* $\chi(G)$.

A lámpák ütemezésének problémája tehát egy gráf kromatikus számának, illetve ennek megfelelő színezésnek a meghatározásához vezetett. Ezzel az úgynevezett *színezési feladattal* később még találkozni fogunk.

Az $\mathcal{A}$ leképezéshez tartozó modelltől azt is elvárjuk, hogy elemei jól ábrázolhatók legyenek számítógépes adatszerkezetekkel. A gráfokra ez teljesül. Egyik természetes ábrázolási módjuk az adjacencia mátrixszal való megadás. Ha a gráf V csúcshalmazának n eleme van, akkor az A adjacencia mátrixa egy n-szer n-es mátrix. Legyen az egyszerűség kedvéért $V = \{1, \ldots, n\}$. Ekkor

$$A[i, j] = \begin{cases} 0 & \text{ha } (i, j) \notin E \\ 1 & \text{ha } (i, j) \in E. \end{cases}$$

Az A mátrixot tekinthetjük kétdimenziós $A[1 : n, 1 : n]$ egész, vagy Boole típusú tömbnek. Modellünk fogalmai ezen az úton könnyen és hajlékonyan ábrázolhatók számítógépes adattípusokkal.

1.1.2. Arthur király civilizációs törekvései

Arthur király fényes udvarában 150 lovag és 150 udvarhölgy él. A király, aki közismert civilizációs erőfeszítéseiről, elhatározza, hogy megházasítja jó lovagjait és szép udvarhölgyeit. Mindezt persze emberségesen szeretné tenni. Csak olyan párok egybekelését akarja, amelyek tagjai kölcsönösen vonzalmat éreznek egymás iránt. Hogyan fogjon hozzá? Természetesen pártfogójához, a nagyhatalmú varázslóhoz, Merlinhez fordul. Merlin rögvest felismeri, hogy itt is bináris szimmetrikus viszonyok ábrázolásáról van szó. Nagy darab pergament vesz elő, és nekilát gráfot rajzolni. A pergamen egyik felén levő U ponthalmaz az udvarhölgyeket jelenti. A másik oldalra kerül a 150 pontú L halmaz; ez pedig a lovagokat ábrázolja. Az l lovagnak megfelelő pont és az u udvarhölgy pontja közé élet rajzol, ha l és u kölcsönösen kedvelik egymást. A modell, amit a mágus használ, a *páros gráf*. Ebben a pontok V halmaza két egymást nem metsző rész egyesítése; él csak a két rész között futhat.

Definíció: *A* $G = (V; E)$ *gráf egy páros (kétrészes) gráf, ha a V csúcshalmaz felbontható két nem üres L és U részre úgy, hogy* $L \cap U = \emptyset$, $L \cup U = V$, *és ha* $(v_1, v_2) \in E$, *akkor a* v_1 *és* v_2 *csúcsok egyike U-ban, a másika pedig L-ben van.*

A királyi parancs teljesítéséhez Merlinnek élek egy olyan rendszerét kell kiválasztania a gráf éleiből, hogy a kiválasztott élek közül a gráf minden pontjához pontosan egy csatlakozzon. A kiválasztott élek felelnek meg a tervezett házasságoknak. A gráfelmélet nyelvén *teljes párosítást* kell keresnie.

Definíció: *A* $G = (L, U; E)$, $\mid L \mid = \mid U \mid = n$ *kétrészes gráf éleinek egy* $S \subseteq E$ *részhalmaza teljes párosítás, ha az* $(L, U; S)$ *gráfban minden pontból pontosan 1 él indul ki.*

Merlinnek tehát a pergamenre rajzolt gráf egy teljes párosítását kellene megadnia. Ha a legenda főváltozatát fogadjuk el hitelesnek (a Sir Thomas Malory által írt *La Morte d'Arthur*t), akkor arra jutunk, hogy a feladatnak nincs megoldása. Ismeretes ugyanis, hogy Arthur és Sir Lancelot is csak és kizárólag Guinevere királyné iránt érez vonzalmat. Hogyan járuljon mármost Merlin a nagyúr elé, aki nem örül túlságosan, ha parancsát nem teljesítik? A király az Excalibur után kapkodó lobbanékonyságán kívül éles elméjéről is nevezetes. Merlin így megmentheti fejét. Annyit kell tennie, hogy megmutatja a rajz megfelelő részletét (amiből kitűnik, hogy Arthur és Lancelot is csak Guinevere-rel van összekötve) a királynak, aki a bizonyíték hatására békésen eláll tervétől.

Általánosabban fogalmazva ugyanez lenne a helyzet, ha tetszőleges k esetén van k lovag úgy, hogy a kedvenceik kevesebb, mint k udvarhölgy közül kerülnek ki. A megfelelő fogalom a gráfok világából a König-akadály. Legyen $G = (L, U; E)$ egy kétrészes gráf, $\mid L \mid = \mid U \mid$. A nem üres $X \subseteq L$ csúcshalmaz egy *König-akadály*, ha van olyan $Y \subseteq U$, $\mid Y \mid < \mid X \mid$, hogy X-ből minden él Y-ba megy. Igazolható, hogy pontosan akkor *nincs* G-ben teljes párosítás, ha van benne König-akadály. Ezt tudva tovább pontosíthatjuk a feladatot: keressünk G-ben teljes párosítást, vagy egy König-akadályt, ha nem létezik teljes párosítás. Az utóbbi esetben világos bizonyítékot kapunk arra, hogy az eredeti feladatnak nincs megoldása. Az $\mathcal{A}$ leképezés ezúttal is segített tisztábbá tenni az eredeti feladatot. A modellel – jelen esetben a gráfokkal – kapcsolatos ismeretek komoly segítséget jelentettek a feladat értelmes és pontos megfogalmazásában.

Feladat: Arthur király kifogyhatatlan az ötletekből, ha a kulturált viselkedést kell előmozdítani Camelot várában. Legutóbb például feltalálta a késsel-villával való étkezés misztériumát. Azt szeretné, ha lovagjai ezentúl nem puszta kézzel nyúlnának a falatokhoz a Kerek Asztal melletti nagy lakomákon. Félő azonban, hogy a nyers modorú bajnokok eme szúró-vágó eszközök birtokában megpróbálják kiegyenlíteni számláikat az asztalszomszédjaikkal. Az ilyen csetepaték elkerüléséhez úgy kellene leültetni a lovagokat, hogy az asztalszomszédok között ne feszüljön ellenséges érzelem. Merlin feladata ezúttal egy olyan, a Kerek Asztal körüli ülésrend készítése, mely szerint senki sem kerül haragosa mellé. Ismert, hogy kik

nem állnak hadilábon egymással (ismét egy bináris szimmetrikus relációval leírható kapcsolat). Próbáljuk gráfok segítségével megfogalmazni a problémát. Milyen objektum keresésére vezet a feladat?

1.2. Algoritmusok

Két példát mutatunk ezután a $\mathcal{B}$ lépésre. Itt már kellően absztrakt, ugyanakkor gépközeli modellben megfogalmazott algoritmikus problémák megoldásáról lesz szó. Először egy irányított gráfokról szóló feladatot veszünk górcső alá.

1.2.1. Szuperforrás keresése

Az előző feladatok irányítatlan gráfokhoz vezettek. Nem tettünk különbséget az (u, v) és a (v, u) pár között. Ha az egyik E-be tartozott, akkor a másik is. Az irányított gráfok abban különböznek ettől, hogy az (u, v) él megléte nem feltétlenül jelenti a (v, u) él meglétét a gráfban. Ennek megfelelően az A adjacencia mátrix nem feltétlenül szimmetrikus, mint az eddigi példákban. Ha egy ilyen gráfot rajzolunk le, akkor az $(u, v) \in E$ élet ábrázoló vonalra v felé mutató nyilat teszünk. Most egy olyan feladattal fogunk foglalkozni, ami már kellően pontos megfogalmazásban áll rendelkezésünkre.

Definíció: *A G irányított gráf $s \in V$ csúcsa szuperforrás, ha minden s-től különböző $y \in V$ csúcs esetén teljesül, hogy $(s, y) \in E$ és $(y, s) \notin E$.*

A szuperforrás olyan s csúcs, amiből a gráf minden más csúcsába él vezet; az s-be pedig egyetlen más csúcsból sem megy él. Nyilvánvaló, hogy egy gráfban legfeljebb egy szuperforrás lehet.

A feladat a következő: *tegyük fel, hogy az A adjacencia mátrixával adott a $G = (V, E)$ irányított gráf, aminek a csúcshalmaza $V = \{1, \ldots, n\}$. Döntsük el, hogy van-e G-ben szuperforrás. Ha igen, találjuk meg.*

Egy megoldás rögtön kínálkozik. Sorra vesszük az $i \in V$ csúcsokat, mindegyikről megnézve, hogy szuperforrás-e. Az i csúcs vizsgálata során az i-be menő és az i-ből kiinduló élek hiányát illetve meglétét kell ellenőrizni. Ez lényegében az A mátrix i-edik sorának és i-edik oszlopának a végignézését jelenti.

Mindjárt felmerül a kérdés, hogy mennyire jó a kapott megoldás. Nem lehetséges-e ennél hatékonyabb, gyorsabb módszer? Hogy erre válaszolni tudjunk, mérnünk kell valahogy a módszereink hatékonyságát, sebességét. Meghatározhatnánk például a programunkban szereplő elemi utasítások végrehajtási idejét, amiből aztán becsülhető lenne a futási idő a különböző n értékekre. Az így kapott

mérőszám egyrészt kissé nehézkes: többféle elemi utasítás idejét kellene kimérni. Másfelől túlságosan gépfüggő is: más környezetben ezek az elemi idők egészen máshogyan alakulhatnak. Ezek a problémák azt sugallják, hogy keressünk valami egyszerű, gép- és környezetfüggetlen mérőszámot. Az algoritmus sebességét mérő szám legyen az *A mátrix megvizsgált elemeinek a száma*.

Első megoldásunkról kiderül, hogy nem valami fényes. Az A mátrix minden főátlón kívüli elemét megnézzük, mégpedig kétszer. Az összköltség $2(n^2 - n)$ ilyen vizsgálat.

A következő módszer azon az egyszerű észrevételen alapul, hogy ha $i \neq j$ esetén $(i, j) \in E$, akkor j nem lehet szuperforrás, $(i, j) \notin E$ pedig i-t zárja ki. Úgy is fogalmazhatunk, hogy $A[i, j]$ értékének ismeretében vagy i, vagy j biztosan kizárható mint lehetséges szuperforrás. Hogy a kettő közül melyik lesz a kizárt csúcs, az függ $A[i, j]$ értékétől, de egyiktől mindenképpen megszabadulhatunk. Ezzel az ötlettel gyorsan tudjuk szűkíteni a tudomásunk szerint még lehetséges szuperforrások körét.

$i := 1, j := n$;
while $i \neq j$ **do**
 if $A[i, j] = 1$ **then** $j := j - 1$

 else $i := i + 1$;
(* *Amikor ideérünk, már csak i lehet szuperforrás,*
 ezt ellenőrizzük a továbbiakban. *)
for $k = 1$ **to** n **do**
 if $k \neq i$ és $(A[i, k] \neq 1$ vagy $A[k, i] \neq 0)$ **then return**(nincs szuperforrás)
return(i szuperforrás).

Az eljárás magától értetődő. Az első ciklus leszűkíti a keresést egyetlen jelöltre. A második ciklus ezt az egy szem jelöltet ellenőrzi. Nézzük most a módszerünk hatékonyságát! Evégből először felső becslést adunk a költségére. A **while**-ciklus végrehajtása során az A tömb $n - 1$ elemét vizsgáljuk meg. Utána már csak az i-edik sor és oszlop elemeit kell néznünk; ez további $2n - 2$ hely. Összesen tehát legfeljebb $3n - 3$ mátrix-elem megtekintését igényli az algoritmus.

Jelölje $T(n)$ a legjobb (leggyorsabb) algoritmus által megvizsgált mátrixelemek számának maximumát az összes n pontú gráfra. Eddigi fejtegetésünk szerint $T(n) \leq 3n - 3$, hiszen *van* egy módszerünk, ami megoldja a feladatot ennél nem több lépésben. Hogy a módszerünket értékelhessük, alsó becslést kell adnunk $T(n)$-re. Ennek segítségével képet kaphatunk arról, hogy eljárásunk teljesítménye mennyire tér el a lehető legjobb módszerétől. Algoritmikus problémák elemzésekor általában igen nehéz érdemi alsó becsléseket nyerni; ezt valamennyire érthe-

tővé teszi az, hogy egy ilyen korlát az összes algoritmusra érvényes megállapítás. A szuperforrás-feladat ilyen szempontból szerencsésnek mondható. Nyilvánvaló, hogy $2n - 2 \leq T(n)$, hiszen ennyi elemet meg kell néznünk pusztán ahhoz, hogy egy adott csúcsról ellenőrizzük, hogy szuperforrás-e.

Ennél élesebb korlátot kaphatunk a következő észrevétellel: 1 él megkérdezése legfeljebb 1 csúcsot zár ki mint lehetséges szuperforrást. Ha ugyanis az $A[i, j]$ vizsgálata előtti tudásunk szerint i és j is lehet még szuperforrás, akkor $A[i, j] = 1$ esetén i, $A[i, j] = 0$ esetén pedig j továbbra is a jelöltek között marad. Az i-től és j-től különböző csúcsok megítélésén a válasz nem változtat.

Arra juthatunk mindebből, hogy a legjobb algoritmus első $n - 2$ kérdése után még legalább két csúcs – mondjuk i és j – lehet szuperforrás. Ez azt jelenti, hogy az A táblázat még nem nézett részének lehet olyan kitöltése, amely szerint i lesz szuperforrás, és olyan is, amelynél j a szuperforrás. Eddig összesen az A mátrix $n - 2$ elemét néztük meg. A megvizsgált elemek közül tehát valamelyik pontunk (mondjuk az i) sorába és oszlopába legfeljebb $(n-2)/2$ esik. Képzeljük el, hogy az „ellenség" úgy tölti ki az A még nem nézett elemeit, hogy i legyen a szuperforrás. Ez is egy lehetséges bemenete (inputja) a feladatnak; a legjobb algoritmusnak erre is működnie kell. Hogy az i szuperforrás-tulajdonságát igazolja, $A[i, i]$ kivételével ismernie kell az i-edik sor és oszlop értékeit. Ez még legalább $2n - 2 - (n - 2)/2$ kérdést jelent. Innen kapjuk, hogy

$$T(n) \geq 2n - 2 - (n - 2)/2 + n - 2 = 3n - 4 - (n - 2)/2.$$

Az érvelés tanúsága szerint legfeljebb $n/2$ kérdés erejéig megközelítettük az optimumot. Elmondhatjuk, hogy gyors és egyszerű algoritmusunk van a feladat megoldására. Az első módszerhez képest a javulás drámai; például ha $n = 500$, akkor az A mátrix 250 000 eleméből legfeljebb csak 1497-et vizsgálunk meg. Megjegyezzük, hogy ennél jobb alsó korlát is adható; ebből kiderül, hogy a módszer nagyon közel van az optimálishoz.

Feladat: Mutassuk meg, hogy $T(n) \geq 3n - 3 - \log_2 n$. (Az ellenség-módszer alkalmazható. Az A mátrix bizonyos elemeinek megkérdezése utáni helyzet *potenciálja* legyen $\sum_i 2^{k_i}$; az összegben azok az i csúcsok szerepelnek, amelyek az adott helyzetbeli ismeretek szerint még lehetnek szuperforrások, k_i pedig az A i-edik sorából és oszlopából már ismert értékek száma. Az ellenség stratégiája: ha lehet, úgy válaszoljon az algoritmus kérdésére, hogy a szuperforrás-jelöltek ne fogyjanak; ha ezt nem teheti, akkor úgy válaszoljon, hogy a potenciál ne nőjön.)

A feladat érdekessége, hogy megoldható volt úgy is, hogy az A adjacencia mátrix n^2 elemének csak egy töredékét néztük meg. Ez a jelenség elég ritka a természetesen felmerülő gráftulajdonságok körében. Legtöbbször az egész mátrixot végig kell néznünk a megoldás során.

Érdemes megfigyelni az alsó becslésnél alkalmazott *ellenség-módszert*. A feltételezett legjobb algoritmus bizonyos kérdéseire adaptívan válaszoltunk: az algoritmus korábbi lépéseitől függően határoztunk meg néhány $A[i, j]$ értéket. Így találtunk egy olyan bemenetet, amely a módszert (viszonylag) sok munkára kényszeríti.

1.2.2. Hosszú egészek párhuzamos összeadása

Ha [a kivonásnál] semmi sem marad, írj egy köröcskét, hogy a hely ne maradjon üresen. A köröcskének kell elfoglalnia ezt a helyet, mert másképp kevesebb hely lenne, és például a másodikat hinnénk elsőnek.

MOHAMED IBN MÚSZA (AL KHVARIZMI) a nulláról.

Itt ismét egy már pontosan megfogalmazott feladatról lesz szó, mégpedig olyanról, amelyet mindenki régóta ismer. Tegyük fel, hogy adottak a decimálisan írt $a_1a_2 \ldots a_n$ és $b_1b_2 \ldots b_n$ n-jegyű természetes számok. Számítsuk ki az összegük $e_0e_1e_2 \ldots e_n$ decimális rendszerbeli alakját. Ez a legelső valamire való algoritmikus probléma, amivel az iskolában találkoztunk. Bizony, rengeteget gyakoroltuk a hindu–arab-algoritmust, amivel jobbról balra haladva kitölthetjük az alábbi táblázat alsó sorát.

$$
\begin{array}{cccccc}
 & a_1 & a_2 & \ldots & a_{n-1} & a_n \\
+ & b_1 & b_2 & \ldots & b_{n-1} & b_n \\
\hline
e_0 & e_1 & e_2 & \ldots & e_{n-1} & e_n
\end{array}
$$

Összesen $2n - 1$ egyjegyű összeadás árán megkapjuk az e_i számjegyeket: mindegyik i-re kiszámoljuk a helyi $a_i + b_i$ összeget (n elemi összeadás), és ehhez még hozzáadjuk az átvitel értékét ($n - 1$ egyjegyű összeadás). A módszer gyors és praktikus. Tulajdonképpen a bemenet hosszával arányos lépésszámban, *lineáris időben* megkapjuk az eredményt. Hasonlót mondhatunk arra az esetre, amikor a műveletet számítógép segítségével végezzük. Ha n viszonylag kicsi, akkor a számaink egy-egy gépi szóban ábrázolhatók, és a feladat egyetlen gépi összeadással megoldható. Ha a számok nagyon hosszúak, akkor a hindu–arab-algoritmus mintájára járhatunk el. A különbség annyi, hogy az elemi lépéseket a gépünk szóhosszától függő méretű többjegyű számokkal végezzük. A műveletek száma így is arányos lesz a bemenet hosszával; az arányossági tényező a szóhossz függvénye. Az ősi módszer tehát hatékony megoldást nyújt mind a kézi, mind pedig a szokásos értelemben vett gépi számoláshoz.

Mit tehetünk akkor, ha a feladat megoldására több processzorunk (számítógépünk) van, amelyek egyidejűen, *párhuzamosan* dolgozhatnak? Tudjuk-e így lényegesen csökkenteni a számolás idejét? Némi kísérletezgetés után rájövünk, hogy a

fő gondot a teendők értelmes szétosztása jelenti. Úgy kellene megszervezni a munkát, hogy a processzorok dolgozhassanak anélkül, hogy túl sokat kelljen várniuk egymásra. Példaképpen nézzük a 843712 és 156288 számok összegét. Megtehetnénk, hogy minden helyiértékhez egy processzort rendelünk, amely elvégzi az ott levő jegyek összeadását. Az utolsót kivéve mindegyik 9-et kap eredményül. Utána várniuk kell a jobb felől érkező átvitelre. A bal szélső processzor csak az összes többi után fejezheti be a munkát; meg kell várnia, amíg a jobb szélről az 1-es átvitel végigballag a processzorokon. Az idő így ismét arányos lesz az összeadandók hosszával, ami azt jelenti, hogy nem értünk el jelentős gyorsulást.

A párhuzamos algoritmusok tervezésében tipikus az itt vázolt helyzet. A megoldáshoz vezető teendőket kell minél inkább párhuzamosítani, függetlenül végezhető részfeladatokra bontani. Vannak olyan feladatok, ahol ez szinte magától értetődő, mint mondjuk a kukoricatörésnél, ahol a munkások külön-külön sorban dolgozva, egymástól függetlenül tevékenykedhetnek. Az ilyen esetekben ideális gyorsulás érhető el; ha t munkás van, a szükséges idő a t-edrészére csökken. Más feladatoknál a párhuzamosítás sokkal nehezebb; úgy találjuk, hogy a processzorok között jelentős egymásra utaltság van. Az országúti kerékpáros csapatversenyt említhetjük példaként. Finoman összehangolt munkával négyen együtt gyorsabban legyűrik a távot, mint azt egy ember tenné. De szó sincs arról, hogy negyedére csökkenne a szükséges idő. Az utóbbi egy nehezebben párhuzamosítható feladat.

Itt most egy erőteljesen párhuzamos megoldást adunk hosszú egészek összeadására. Mint látni fogjuk, lesz egy kis huncutság a dologban. A szélvészgyors megoldás ára az, hogy az eredményt nem a megszokott formában kapjuk.

Definíció: *A* $\sum_{i=1}^{n} a_i 10^{n-i}$, $0 \le a_i \le 9$ *alakú számábrázolást* standard *ábrázolásnak nevezzük. A* $\sum_{i=1}^{n} a_i 10^{n-i}$, $-9 \le a_i \le 9$ *alakú számábrázolást pedig* kiegyensúlyozottnak *nevezzük.*

A standard ábrázolás tehát a szokásos decimális felírást jelenti, ahol a megengedett számjegyek a $0, 1, 2, \ldots, 9$. A kiegyensúlyozott ábrázolásban tizenkilenc *jegyet* használhatunk. Ezek: $-9, -8, \ldots, -1, 0, 1, \ldots, 8, 9$. Tudjuk, hogy a standard alakban való felírás egyértelmű. Az így ábrázolt számokról könnyű megállapítani, hogy melyik a nagyobb. A kiegyensúlyozott ábrázolás nem egyértelmű, az ilyen számok összehasonlítása elég körülményes. A többértelmű felírásnak van viszont egy olyan előnye, ami segíteni fog abban, hogy megállítsuk az átvitel hosszú futását az összeadásnál.

Állítás: *Tetszőleges* $-18 \le a \le 18$ *egésznek van olyan kiegyensúlyozott ábrázolása, melyben nincs* $+9$*-es és* -9*-es jegy, továbbá a 10-es helyiértékű jegy -1,0 vagy 1.*

Bizonyítás: Ha $a \neq 9$ és $a \neq -9$, akkor a standard alak megfelelő. A kimaradó két számnál a $9 = 10 - 1$ és $-9 = -10 + 1$ egyenlőségeket használhatjuk. A kiegyensúlyozott felírások ekkor $1 - 1$, illetve -11. □

Ezután ismertetjük *Avizienis algoritmusát* két n-jegyű kiegyensúlyozott felírású szám összeadására. Az $a_1 \ldots a_n$ és $b_1 \ldots b_n$ n-jegyű egészek $e_0 e_1 \ldots e_n$ kiegyensúlyozott alakú összegét fogjuk megkapni. Az algoritmus n processzort használ, ezek közül az i-edik processzor az i helyiértékű jegyeken dolgozik. Az első lépésben párhuzamosan kiszámítják az a_i+b_i összeg egy olyan (kétjegyű) $c_i d_i$ kiegyensúlyozott reprezentációját, amilyet az állítás szavatol: $a_i + b_i = c_i 10 + d_i$, ahol $c_i \in \{-1, 0, 1\}$ és $-8 \leq d_i \leq 8$. A második lépésben csak az 1. és n. processzorok dolgoznak, megadva a végeredmény első és utolsó jegyét: $e_0 = c_1$, $e_n = d_n$. A harmadik, egyben utolsó lépésben az i-edik processzor $1 \leq i \leq n - 1$ egyetlen összeadással megkapja e_i-t: $e_i = d_i + c_{i+1}$. A módszer helyessége abból adódik, hogy az $i+2, \ldots, n$ helyiértékeknél kapott eredmények nem befolyásolják az $i + 1$. helyről az i-edikre menő átvitel értékét. A d_{i+1} jegy ugyanis -8 és 8 közé esik, amit a jobbról jövő -1, 0, vagy 1 átvitel nem tud 9 fölé, vagy -9 alá vinni. Az első lépésben számított c_{i+1} érték tényleg a helyes átvitel, amit az i. helyen figyelembe kell vennünk. A számítás menete így szemléltethető:

		a_1	…	a_{n-1}	a_n
		b_1	…	b_{n-1}	b_n
1. lépés:		$c_1 d_1 := a_1 + b_1$	…	$c_{n-1} d_{n-1} := a_{n-1} + b_{n-1}$	$c_n d_n := a_n + b_n$
2. lépés:	$e_0 := c_1$				$e_n := d_n$
3. lépés:		$e_1 := d_1 + c_2$	…	$e_{n-1} := d_{n-1} + c_n$	

Nézzük meg, hogy fest ez egy konkrét példán. A korábban problémásnak minősült 843712 és 156288 számokon követjük a módszer lépéseit. Csak a kapott részeredményeket tüntetjük fel.

		8	4	3	7	1	2
		1	5	6	2	8	8
1. lépés:		$1-1$	$1-1$	$1-1$	$1-1$	$1-1$	10
2. lépés:	1						0
3. lépés:		0	0	0	0	0	
Összeg:	1	0	0	0	0	0	0

Avizienis algoritmusával tehát konstans párhuzamos időben megkaptuk az eredmény egy kiegyensúlyozott reprezentációját. Ezzel a kukoricatörésnél tapasztalthoz hasonló ideális felgyorsítást sikerült elérni.

Egy n-jegyű kiegyensúlyozott szám standard alakra hozására csak viszonylag lassúnak mondható, $c \log n$ párhuzamos lépést igénylő módszer ismert. Ezt

használva $O(\log n)$ párhuzamos idejű módszer adódik *két* n-jegyű egész standard felírású összegének a meghatározására. Avizienis módszere különösen akkor hasznos, ha *sok* összeadást kell elvégeznünk egyszerre. Erre a helyzetre értelmes példát kínál a szorzás.

Feladat: Mutassuk meg, hogy két n-jegyű egész szorzása elvégezhető $O(\log n)$ párhuzamos időben. (Az iskolában tanult szorzóalgoritmus összeadásait Avizienis módszerével végezhetjük el; csak a végeredményt írjuk át standard formába.)

A módszer kulcsa a 9-es nélküli felírhatóságról szóló állítás. Ez kézenfekvően általánosítható tetszőleges $k > 2$ alapú számrendszerre; ekkor a 9-es szerepét $k-1$ játssza. A gyakorlatban a négyes számrendszert használják, mert a gépeken szokásos kettes számrendszer és a négyes számrendszer közötti átalakítások gyorsak.

Feladat: Adjunk konstans párhuzamos idejű módszert kettes számrendszerben adott számok négyes számrendszerbeli átírására, és a fordított irányú átalakításra.

A hosszú egészek összeadásának feladatával azt is szerettük volna érzékeltetni, hogy a megoldás hatékonyságáról való képünk függ attól is, hogy milyen számítógépes eszközök állnak a rendelkezésünkre. A megoldás keresése során ezért figyelemmel kell lennünk erre a háttérre. Úgy is fogalmazhatunk, hogy az eszköz-háttér alapvető jellemzői (mint pl. a processzorok száma) részét képezik a modellnek, amivel dolgozunk.

2.

Rendezés

...egész nap a barlangban szorgoskodtam, ahol a vert pénzt kellett kétszersültes zsákokba töltenem... azt hiszem, sohasem szórakoztam jobban, mint amikor ezeket rendezgettem. Angol, francia, spanyol, portugál aranyak, György és Lajos tallérok, dublónok, kettős guineák, monédorok és zecchinók. ROBERT LOUIS STEVENSON
(Jim emlékei a Kincses Szigetről)

Ebben a fejezetben az egyik klasszikusnak számító, alapvető számítási feladattal, a rendezéssel foglalkozunk. Ennek értelmezéséhez nézzük először a rendezési reláció fogalmát. Legyen U egy halmaz, és $<$ egy kétváltozós reláció U-n. Ha $a, b \in U$ és $a < b$, akkor azt mondjuk, hogy a kisebb, mint b. A $<$ reláció egy *rendezés*, ha teljesülnek a következők:

1. $a \not< a$ minden $a \in U$ elemre ($<$ irreflexív);
2. Ha $a, b, c \in U$, $a < b$, és $b < c$, akkor $a < c$ ($<$ tranzitív);
3. Tetszőleges $a \neq b \in U$ elemekre vagy $a < b$, vagy $b < a$ fennáll ($<$ teljes).

Ha $<$ egy rendezés U-n, akkor az $(U, <)$ párt *rendezett halmaznak* nevezzük. Ha az U halmaz egy számítógépes adattípus is egyben, akkor az $(U, <)$ párt *rendezett típusnak* is nevezik. Különösen ez utóbbiak érdekesek a számunkra. Az U elemeit ekkor adatoknak tekinthetjük, és feltehetjük, hogy a $<$ reláció hatékonyan a rendelkezésünkre áll: ha adottak az $a, b \in U$ elemek, akkor az $a < b$ viszony ellenőrzése, más szóval a és b *összehasonlítása* hatékonyan megtehető. Ezt az összehasonlító eljárást gyakran a rendszer biztosítja, olykor viszont magunknak kell megírnunk.

Példák:

1. $\mathbb{Z}$ az egész számok halmaza. A $<$ rendezés a nagyság szerinti rendezés. $(\mathbb{Z}, <)$ tekinthető rendezett típusnak is.

2. Az *abc* betűinek Σ halmaza; a $<$ rendezést az *abc-sorrend* adja. Az x betű kisebb, mint az y betű, ha x előbb szerepel az *abc*-sorrendben, mint y.
3. A Σ betűiből alkotott szavak Σ^* halmaza a szótárszerű vagy *lexikografikus* rendezéssel. Ezt a következő recepttel értelmezhetjük: legyen $X = x_1x_2\cdots x_k$ és $Y = y_1y_2\cdots y_l$ $(x_i, y_j \in \Sigma)$ két szó (X az x_i, Y pedig az y_j betűk sorozata). Az X kisebb mint Y, ha vagy $l > k$ és $x_i = y_i$ ha $i = 1, 2, \ldots, k$; vagy pedig $x_j < y_j$ teljesül a legkisebb olyan j indexre, melyre $x_j \neq y_j$. Tehát például $kar < karika$ és $bor < bot$.

A rendezés feladata általánosan így fogalmazható: *adott az* $(U, <)$ *rendezett halmaz elemeinek egy* $u_1, u_2, \ldots, u_n$ *sorozata; rendezzük ezt át egy nem csökkenő* $v_1, v_2, \ldots, v_n$ *sorrendbe.*

A nem csökkenő sorrend azt jelenti, hogy $v_1 \leq v_2 \leq \ldots \leq v_n$ teljesül, ahol $\leq$ a kisebb vagy egyenlő reláció jelölése. Az egyenlőséget akkor kell megengednünk, ha az u_i sorozatban ismétlődések vannak, ami az alkalmazásoknál gyakran előfordul.

Az $u_1, u_2, \ldots, u_n$ sorozat többféle módon adható meg. Kaphatjuk tömb, láncolt lista formájában vagy egy külső táron levő állomány rekordjainak sorozataként. Legtöbbször a $v_1, v_2, \ldots, v_n$ sorozatot ugyanolyan formában követelik tőlünk, mint amilyen a bemenet volt. Ezért a rendező módszerek átalakító lépései többnyire az elemek (tömbelem, listaelem, rekord) *mozgatásai*, *cseréi*.

A rendezési feladat két szempontból fontos. Gyakran előfordul, hogy magában a kérdés megfogalmazásában szerepel a rendezés. Például, ha arra vagyunk kíváncsiak, hogy egy nyilvántartásban szereplő személyek közül kinek a legnagyobb a fizetése, vagy hogy kiknek a fizetése van az átlag fölött stb. A másik, talán még fontosabb alkalmazás a rendezés használata a keresés megkönnyítésére. Ezzel később behatóan foglalkozunk. Most csak egy példát említünk. A telefonkönyvhöz általában azért fordulunk, hogy (gyorsan) megtaláljunk egy telefonszámot. Nem érdekel bennünket különösebben a nevek rendezése, de a név szerinti rendezettség komoly segítséget nyújt a gyors keresésben (képzeljük el, mi lenne, ha a telefonkönyv bejegyzéseit csak úgy, össze-vissza vetnék papírra).

A fejezet elején a rendezett halmazokban való kereséssel foglalkozunk. Utána áttekintjük a legfontosabb összehasonlítás alapú rendezéseket. Az ilyen módszerek a döntéseiket kizárólag az elemek közötti $<$ relációra alapozzák. A kulcsmanipulációs rendezések ezzel szemben használhatják az elemek (kulcsok) belső szerkezetének ismeretéből eredő előnyöket. Megismerünk két ilyen szemléletű eljárást is. Ezután bemutatunk egy gyors párhuzamos rendező módszert, végül pedig külső tárakon levő állományok rendezésével foglalkozunk.

2.1. Keresés rendezett halmazban

A rendezett halmazokban való keresési eljárások jelentős szerepet játszanak a legkülönfélébb adatkezelő rendszerekben. Érdemes tehát közelebbről is megismerkedni velük. Tegyük fel, hogy adott az $(U, <)$ rendezett halmaz véges

$$S = \{s_1 < s_2 < \ldots < s_{n-1} < s_n\}$$

részhalmaza. Az S-re úgy gondolhatunk, mint az U általunk tárolt elemeinek összességére. Adott ezenkívül a bemenet részeként egy $s \in U$ elem. A feladatunk annak eldöntése, hogy $s \in S$ teljesül-e. Igenlő válasz esetén általában az s elem S-beli „helye" is érdekel bennünket.

A feladatot összehasonlítások segítségével szeretnénk megoldani. Egy ilyen lépésben vesszük valamelyik s_i-t (S-nek a rendezés szerinti i-edik elemét), és összehasonlítjuk az s elemmel. Az eredmény lehet $s = s_i$, $s_i > s$ vagy $s > s_i$. Az eredménytől függően további ilyen összehasonlításokat végzünk, egészen addig, amíg az ($s \in S$?) kérdést meg nem tudjuk válaszolni. A fontosabb módszerek a következők.

Lineáris keresés

Az s-et először az S halmaz legkisebb elemével, s_1-gyel hasonlítjuk össze. Ha $s < s_1$, akkor végeztünk, megállapítva, hogy s nincs S-ben. Ha $s = s_1$, akkor sikerrel jártunk: az ($s \in S$?) kérdésre a válasz igenlő. Ha pedig $s > s_1$, akkor továbbmegyünk, és az s_2 elemet vetjük össze s-sel. Ennek a kimeneteleit az előzőhöz hasonlóan kezeljük. Vagy választ kapunk a keresőkérdésre, vagy folytatjuk s_3-mal. És így tovább. Addig lépkedünk előre az $s_1, s_2, \ldots, s_i, \ldots$ sorozat mentén, amíg az ($s \in S$?) kérdés el nem dől.

Mi mondható a módszer – összehasonlításokban mért – költségéről? A legkedvezőtlenebb esetben, amikor is $s \geq s_n$, az S halmaz minden elemével hasonlítunk; ez $|S| = n$ összehasonlítást jelent. Ennél a feladatnál szokás *átlagos lépésszámról* is beszélni. Ekkor azzal a feltevéssel élünk, hogy a keresett s elem egyenlő eséllyel lehet a $(-\infty, s_1]$, $(s_1, s_2], \ldots, (s_{n-1}, s_n]$ és (s_n, ∞) intervallumok bármelyikében. Itt $(a, b]$ az U azon s elemeinek az összességét jelenti, melyekre $a < s \leq b$ igaz. Hasonlóan, $(-\infty, b]$ az U halmaz b-nél nem nagyobb, (a, ∞) pedig az a-nál nagyobb elemeinek halmaza.

Ha $s \in (-\infty, s_1]$, akkor 1 összehasonlítást végzünk, ha pedig $s \in (s_i, s_{i+1}]$ akkor $i + 1$-et $(1 \leq i < n)$. Az utolsó intervallum $(s_n, \infty]$ elemeire a költség n összehasonlítás. A kapott $n + 1$ szám átlaga

$$\frac{1 + 2 + \cdots + n - 1 + n + n}{n + 1} = \frac{n(n+1)}{2(n+1)} + \frac{n}{n+1} = \frac{n}{2} + \frac{n}{n+1}.$$

A módszer átlagos költsége tehát $(n/2) + 1$ összehasonlítás körül van.

Bináris keresés

Az eljárás az *oszd meg és uralkodj* elven alapul. A feladat megoldását sokkal kisebb hasonló feladatok megoldására egyszerűsítjük. Először az S halmaz középső elemével hasonlítjuk össze s-et. Ennek az s_i elemnek az i sorszáma lehet k vagy $k + 1$, ha $n = 2k$. Ha pedig $n = 2k + 1$, akkor $i = k + 1$. Mit ad az s és s_i összevetése? Ha $s = s_i$, akkor egyetlen lépésben végeztünk. Ha $s < s_i$, akkor az s elemet elég már csak az $\{s_1, \ldots, s_{i-1}\}$ részhalmazban keresni. Ugyanígy, ha $s > s_i$, akkor az $\{s_{i+1}, \ldots, s_n\}$ részhalmazra szorítkozhatunk. Egyetlen jól irányzott kérdéssel tehát vagy megtaláljuk az s elemet, vagy pedig visszavezetjük a kérdést egy sokkal kisebb S_1 rendezett halmazban való keresésre. Az S_1 mérete legfeljebb az S méretének a fele: $|S_1| \leq |S|/2 = n/2$. Az eljárást ismételjük, amíg ez a felezés lehetséges. Az egyre zsugorodó célhalmazok legyenek

$$S_0 = S, S_1, \ldots, S_j.$$

Az utolsó S_j halmazról feltehetjük, hogy nem üres, más szóval $1 \leq |S_j|$, továbbá, hogy az S_j középső elemével való összehasonlítás már megválaszolja az ($s \in S$?) keresőkérdést. Eszerint a felhasznált összehasonlítások száma $j + 1$. Az $|S_{i+1}| \leq |S_i|/2$ egyenlőtlenségek összefűzésével kapjuk, hogy $|S| = n$, $|S_1| \leq n/2$, $|S_2| \leq n/4$, ... ,$|S_j| \leq n/2^j$. Az utolsó szerint $1 \leq n/2^j$, amiből $j \leq \log_2 n$.

Összesen tehát $j + 1 \leq \log_2 n + 1$ összehasonlító lépés elég. Picivel pontosabb a $j + 1 \leq \lceil \log_2(n + 1) \rceil$ korlát. Ez $j \leq \lfloor \log_2 n \rfloor < \log_2(n + 1)$ miatt érvényes.

Feladat: Mutassuk meg, hogy a bináris keresés optimális: kevesebb összehasonlítással nem oldható meg a feladat. (Alkalmazzuk az ellenség-módszert. Az ellenség úgy válaszol az algoritmus kérdéseire, hogy utána s keresését a nagyobbik részhalmazban kell folytatni.)

A két módszert, a lineáris keresést és a bináris keresést egybevetve megállapíthatjuk, hogy az utóbbi sokkal kevesebb összehasonlítást igényel. Mégsem mondhatjuk, hogy a lineáris keresés rossz módszer. Ha ugyanis az S halmazt listaként vagy külső táras állomány részeként kapjuk, akkor a bináris keresés többnyire nem alkalmazható. A gond ott van, hogy ekkor nem tudunk elég gyorsan a halmaz közepébe ugrani. A lineáris lépkedés pedig a legszerényebb szervezési módok mellett is megy. A tömbök fontos előnye, hogy alkalmasak a bináris keresésre. Általában a programozási környezet biztosítja, hogy az $A[1 : n]$ tömb bármelyik $A[i]$ eleme gyorsan elérhető.

Az itt tárgyalt keresési feladatnak több értelmes változata van. Például az S lehet egy rendezett sorozat. A különbség annyi, hogy ekkor az s_i elemek között

ismétlődések is előfordulhatnak. A módszerek költségéről mondottak ekkor is érvényben maradnak, legalábbis ami a legrosszabb eseteket illeti.

Egy másik érdekes változat, amikor a keresett s elem összes S-beli előfordulását meg kell találnunk. A költség ekkor függ az ismétlődések számától is.

Példa: Tegyük fel, hogy az $A[1:n]$ tömbben

név	*cím*	*telefonszám*

alakú bejegyzéseket tárolunk, a *név* szerint rendezve. A rendezettség annyit tesz, hogy ha $i < j$, akkor az $A[i]$-ben levő *név* mező értéke nem nagyobb, mint az $A[j]$-ben levőé. Ha mondjuk Rezeda Kázmér telefonszámára vagyunk kíváncsiak, akkor bináris keresést használhatunk. Az A tömb legfeljebb $\lceil \log_2(n+1) \rceil$ elemének a megnézésével kiderül, hogy van-e információnk ilyen nevű személyről. Ha igen, akkor mindjárt kapunk is egy Rezeda Kázmérról szóló bejegyzést. Ha az összes Rezeda Kázmérra kíváncsiak vagyunk, akkor ettől a találati helytől jobbra, illetve balra lépkedve megtaláljuk valamennyit. Ez még körülbelül annyi tömbelem elérését jelenti, mint ahányszor ismétlődik a név.

Megemlítünk még egy rendezett halmazban való keresési módszert. Ez az ún. intelligens kereső rendszerekben fordul elő, használata elég bonyolult szervezést igényel. A neve *interpolációs keresés*. Csak a módszer alapgondolatát szeretnénk itt érzékeltetni. Amikor a telefonkönyvben például Fátyol Szilvia számát keressük, akkor nem fogunk az elejétől kezdve lineárisan lépkedni. Nem próbálkozunk a telefonkönyv közepével sem, a bináris keresésnek megfelelően. A kötetet inkább valahol az első negyede táján nyitjuk ki, mert úgy érezzük, hogy az F-fel kezdődő nevek arrafelé vannak. Az F-betűs neveknél pedig nagy léptekkel igyekszünk megtalálni az elsőket, hiszen az á az abc elejéről való.

Arról van itt szó, hogy információval rendelkezünk a tárolt halmaz elemeinek az eloszlásáról, aminek a segítségével jól-rosszul becsülni tudjuk a keresett elem lehetséges helyét. Az interpolációs keresők ilyen természetű ismeretek felhasználásával próbálják gyorsítani a keresést. Az információ minőségétől függően az összehasonlítások száma akár $\log\log n$-re is levihető.

2.2. Összehasonlítás alapú rendező módszerek

Itt az olyan rendező eljárások legfontosabbjait érintjük, amelyek nem tekinthetnek a rendezendő elemek (másik szokásos szóhasználattal: *kulcsok*) belsejébe. Az elemekkel kapcsolatos döntéseiket csak és kizárólag a $<$ relációra alapozhatják. Ezeket összehasonlítás alapú módszereknek nevezzük. Az eljárásokat tömbként

megadott bemenetek esetére ismertetjük. Az olvasóra bízzuk annak meggondolását, hogy az elmondottak miként és mennyire maradnak érvényben más megadási módokra. A feladat ezek után a következő:

Adott az $(U, <)$ rendezett halmazból (típusból) való elemekből álló $A[1:n]$ tömb. Rendezzük át az A elemeit úgy, hogy azok a $<$ szerint nem csökkenő sorrendben legyenek.

A követelmény szerint az eljárás végén az A-nak ugyanazokat az elemeket kell tartalmaznia és ugyanakkora multiplicitással, mint kezdetben. A végállapotban érvényesek az $A[1] \leq A[2] \leq \ldots \leq A[n]$ egyenlőtlenségek. A módszerek költségének számításakor általában két tényezőt veszünk figyelembe: az összehasonlításokat és az elemek mozgatásait[1].

2.2.1. Buborék-rendezés

A rendezésére egy kézenfekvő ötlet, hogy a rosszul álló szomszédos elempárokat megcseréljük: ha valamely i-re $A[i] > A[i+1]$, akkor a két cella tartalmát kicseréljük. Ha már nincs ilyen értelemben rosszul álló pár, akkor A rendezett állapotban van. Az ötlet egy kulturált kivitelezése a *buborék-rendezés*. A tömb elejéről indulva a rosszul álló szomszédos elemeket cserélgetve eljutunk a tömb végéig. Ekkor a legnagyobb elem(ek egyike) $A[n]$-ben van. Utána ismételjük ezt az $A[1:n-1]$ tömbre, majd az $A[1:n-2]$ tömbre, stb. Végül A rendezett lesz.

procedure buborék
(* *az* $A[1:n]$ *tömböt nem csökkenően rendezi* *)
for $(j = n-1, j > 0, j := j-1)$ **do**
 for $(i = 1, i \leq j, i := i+1)$ **do**
 { ha $A[i+1] < A[i]$, akkor cseréljük ki őket.}

A külső ciklus j változója vezérli az éppen aktuális $A[1:j+1]$ tömb méretét. A belső ciklusban pedig végigmegyünk ezen a résztömbön. Az elvégzett összehasonlítások száma ennek megfelelően $n-1, n-2, \ldots, 2, 1$. Ez összesen $\frac{n(n-1)}{2}$ összehasonlítást jelent minden bemenetre. A legrosszabb esetben ugyanennyi az elemcserék száma is. Ez akkor fordulhat elő, ha kezdetben a tömb fordítottan rendezett: $A[1] \geq A[2] \geq \ldots \geq A[n]$.

[1]Feltevésünkkel elhanyagolunk néhány további költségtényezőt. Ilyen például a ciklusváltozók léptetésének az ideje. Ezek azonban nem befolyásolják az összköltség *nagyságrendjét*.

A módszer nevének magyarázatához képzeljük el, hogy a tömb függőleges helyzetben áll, $A[n]$ van legfelül és $A[1]$ legalul. Képzeljük el még azt is, hogy a nagyobb elemek könnyebbek a kisebbeknél. A módszer alkalmazásakor a könnyebb elemek felfelé mozognak, mint a buborékok valami folyadékban.

Az eljárás, mint később látni fogjuk, a versenytársainál sokkal több összehasonlítást használ, ha n nagy. A kódja viszont rövid, egyszerű, és könnyen átírható tömb helyett listával megadott bemenetre. Eme tulajdonságai miatt kis sorozatok (maximum 10-20 eleműek) rendezésére ajánlható. A módszer *stabil* (másik szokásos szóval: *konzervatív*) abban az értelemben, hogy az egyforma elemek egymás közti sorrendjén nem változtat.

2.2.2. Beszúrásos rendezés

A módszer alapgondolata igen egyszerű. Tegyük fel, hogy az $A[1:k]$ résztömb már rendezett. Ezt használva rendezzük az $A[1:k+1]$ résztömböt. Kezdetben, amikor is $k = 1$, a feltétel nyilván teljesül. Ezután sorra léptetve k-t rendezzük az $A[1:2]$, $A[1:3]$, …, $A[1:n-1]$, $A[1:n]$ tömböket. Ezzel végül megoldjuk a feladatot.

Hogyan jutunk k-ról $k+1$-re? Meg kell találnunk az $A[k+1]$ elem helyét az $A[1:k]$ rendezett résztömb elemei között. Ezt a *beszúrásnak* nevezett lépést egy példán keresztül mutatjuk be. Tegyük fel, hogy $k = 4$ és az $A[1:5]$ résztömb az alábbi:

1	5	7	9	6

Az $A[1:4]$ résztömb már rendezett. Ebben megkeressük az utolsó elemnek – ami esetünkben 6 – a rendezett sorrend szerinti helyét. Ezt tehetjük lineáris vagy bináris kereséssel. A keresés eredményeként kiderül, hogy a 6 helye az $A[2]$ és az $A[3]$ elemek között van. A beszúrást úgy végezhetjük, hogy az $A[3:4]$ résztömb elemeit eggyel jobbra toljuk, és a hatost az így szabaddá tett $A[3]$-ba mozgatjuk. Ezzel az $A[1:5]$ résztömböt rendeztük:

1	5	6	7	9

Nézzük meg a módszer költségét! Ez bizonyos mértékig függ a beszúró lépésben alkalmazott keresési eljárástól.

Beszúrásos rendezés lineáris kereséssel

Ami az összehasonlítások számát illeti, a legkedvezőtlenebb esetet akkor kapjuk, amikor az A tömb már a bemeneti állapotában rendezett. A $k+1$-edik

elem beszúrásakor ebben az esetben k összehasonlítást használunk. Ezeket $k = 1, 2, \ldots, n-1$-re összeadva $\frac{n(n-1)}{2}$ összehasonlítás adódik. Könnyű meggondolni, hogy a cserék szempontjából a fordítottan rendezett tömb adja a legrosszabb esetet. A $k+1$-edik elem beillesztéséhez $A[1:k+1]$ mindegyik elemét mozgatni kell. Összesen ez $\frac{(n+2)(n-1)}{2}$ mozgatást jelent.

Vizsgálhatjuk a módszer átlagos viselkedését is. Hogy ezt megtehessük, meg kell mondanunk, hogy milyen lehetséges bemenetekre számoljuk az átlagot. Nos, tegyük fel, hogy az átlagot a tömbnek az $1, 2, \ldots n-1, n$ számokkal való összes lehetséges kitöltésére vesszük. Itt természetesen nincs jelentősége annak, hogy mik a tömbelemek, hiszen a módszer csak a $<$ relációt használja. Mindössze az számít, hogy n különböző elem, és ezáltal $n!$ lehetséges input sorrend van.

Feladat: Tegyük fel, hogy $A[1:k]$ már rendezett, és az $A[k+1]$ elemmel még nem foglalkoztunk. Mutassuk meg, hogy a beszúrás során $A[k+1]$ ugyanannyiszor eshet az első k elem által meghatározott $k+1$ intervallum bármelyikébe. (Legyen H az $\{1, 2, \ldots, n\}$ egy $k+1$-elemű részhalmaza, és nézzük azokat az inputokat, amelyeknél az első $k+1$ elem pont a H-ból van. Ezek között mindegyik $i \in H$ ugyanannyiszor lesz a $k+1$. helyen.)

A feladat szerint teljesül a feltétel, amit a lineáris keresés átlagos viselkedésének elemzésekor megköveteltünk. Az összehasonlítások átlagos száma tehát a k-ról $k+1$-re lépésnél legalább $\frac{k}{2} + \frac{k}{k+1} > \frac{k}{2}$. Ezt összegezve ($k = 1, 2, \ldots, n-1$) kiderül, hogy az eljárás átlagosan legalább $\frac{n(n-1)}{4}$ összehasonlítást végez. Lényegében ugyanezen érvelés mutatja, hogy a mozgatások átlagos száma is $n^2/4$ körül van.

Beszúrásos rendezés bináris kereséssel

A már rendezett k elem közé a beszúrás $\lceil \log_2(k+1) \rceil$ összehasonlítást igényel. Ez a $k = 1, \ldots, n-1$ értékekre összesen

$$(*) \qquad K := \lceil \log_2 2 \rceil + \cdots + \lceil \log_2 n \rceil$$

összehasonlítást jelent. Innen tetszőleges n-re $K \leq n \lceil \log_2 n \rceil$. A K költséget a $(*)$ kifejezés alapján másként, valamivel pontosabban is becsülhetjük. Világos, hogy $\lceil \log_2 k \rceil < 1 + \log_2 k$, amiből

$$K < n - 1 + \log_2 2 + \cdots + \log_2 n = n - 1 + \log_2(n!).$$

Most segítségül hívjuk a Stirling-formulát, ami a faktoriális-függvény növekedési rendjét fejezi ki elemi függvényekkel. A nevezetes formula következő alakját

használjuk: $n! \sim (n/e)^n \sqrt{2\pi n}$. Logaritmusra térve

$$\log_2 n! \sim n(\log_2 n - \log_2 e) + \frac{1}{2}\log_2 n + \log_2 \sqrt{2\pi} \leq n(\log_2 n - 1,442)$$

adódik, feltéve, hogy n elég nagy. Azt is használtuk itt, hogy $\log_2 e = 1,442695\ldots$, és $\log_2 \sqrt{2\pi} = 1,32\ldots$. Kapjuk, hogy $K \leq n(\log_2 n - 0,442)$ elég nagy n-re. Az összehasonlítások száma sokkal kedvezőbb, mint a lineáris keresést használó változaté vagy a buborék-rendezésé. Rövidesen látni fogjuk, hogy ebből a szempontból a bináris keresést használó módszer közel optimális.

Ha viszont a mozgatásokat is számításba vesszük, akkor ez a módszer sem bizonyul túl jónak. A mozgatások számát illetően a lineáris változatról mondottak itt is érvényesek, hiszen a mozgatások száma nem függ a keresési módszertől. A legrosszabb esetben $\frac{(n+2)(n-1)}{2}$, átlagosan pedig mintegy $\frac{n^2}{4}$ mozgatást végzünk.

2.2.3. Egy alsó becslés

Itt egy egyszerű, általános alsó becslést adunk rendező módszerek összehasonlításainak számára. A korlát érvényes lesz minden összehasonlítás alapú eljárásra.

Tegyük fel, hogy van egy rendező algoritmusunk, ami helyesen rendezi az $(U, <)$ rendezett halmaz rögzített $u_1 < u_2 < \ldots < u_n$ elemének minden input permutációját. Ez azt jelenti, hogy az u_i elemek mind az $n!$ lehetséges bemeneti sorrendjét cserékkel átalakítja az $u_1, u_2, \ldots, u_n$ sorrendre. Tegyük még fel, hogy a módszer kizárólag két kimenetelű döntéseken alapul. Ezen azt értjük, hogy felírható, mint

$$\textbf{if } \textit{feltétel} \textbf{ then } \textit{akció}_1 \textbf{ else } \textit{akció}_2 \text{ ;}$$

alakú utasítások egymásutánja. Itt $\textit{akció}_i$ két dolgot jelenthet: egy másik utasításra való ugrást vagy pedig egy átrendezést. Az átrendezés az elemek akármilyen bonyolult csereberéje lehet.

Tegyük fel, hogy a módszer legfeljebb k döntéssel megoldja a feladatot az $u_1, u_2, \ldots, u_n$ elemek $n!$ lehetséges bemeneti sorrendjének bármelyikére. Megmutatjuk, hogy ekkor $k \geq \log_2(n!)$.

Evégből nézzük a módszer működését a $v_1, v_2, \ldots, v_n$ input sorozaton, és jegyezzük fel sorban a kapott döntések eredményeit. Ezáltal egy legfeljebb k hosszúságú, *igen*, *nem* értékekből álló sorozatot, *kódszót* kapunk.

Azt állítjuk, hogy így különböző bemeneti sorozatokhoz különböző kódszavakat rendelünk. Legyen ugyanis $v_1, v_2, \ldots, v_n$ és $w_1, w_2, \ldots, w_n$ két bemenet, amelyekhez ugyanaz a válasz-sorozat tartozik. Ekkor ugyanaz lesz a végrehajtásra kerülő átrendezések $P_1, P_2, \ldots, P_s$ sorozata is. Végül mindkét esetben a rendezett $u_1, u_2, \ldots, u_n$ sorrendet kapjuk. A P_i akciók *invertálhatók*, hiszen a

leképezés, amit megvalósítanak, kölcsönösen egyértelmű. Léteznek tehát a P_i^{-1} „visszarendezések". Ezért ha a $v_1, v_2, \ldots, v_n$ bemeneten a $P_1, P_2, \ldots, P_s$ átrendezések az $u_1, u_2, \ldots, u_n$ sorozathoz vezettek, akkor az utóbbira alkalmazva a $P_s^{-1}, P_{s-1}^{-1}, \ldots, P_1^{-1}$ átrendezéseket visszakapjuk a $v_1, v_2, \ldots, v_n$ sorozatot. Ugyanez elmondható a $w_1, w_2, \ldots, w_n$ bemenetre is. A két bemenet tehát megegyezik.

Ha egy bemenethez tartozó válaszsorozat k-nál rövidebb, akkor megfelelő számú *igen*t a végére írva egészítsük ki k hosszúságúra. Ezután is igaz marad, hogy különböző bemenetekhez különböző kódszavak tartoznak. Ez azon múlik, hogy egy (eredeti értelemben vett) kódszó nem lehet egy másik folytatása. Ugyanis amíg a válaszok egyeznek, addig pontosan ugyanazokat az utasításokat hajtjuk végre mindkét esetben. Ha a két kódszó az első j válaszig egyezik, és az egyik bemenetnél a program végére értünk, akkor a másiknál is ott leszünk.

Az eddigieket összegezve arra jutottunk, hogy az $n!$ lehetséges input mindegyikéhez egy k hosszúságú kódszót rendeltünk, mégpedig különböző bemenetekhez különböző kódszavakat. A kódszavak száma tehát legalább annyi, mint a bemenetek száma. A szóbajövő kódszavak száma 2^k, amiből $2^k \geq n!$, és végül $k \geq \log_2(n!)$ adódik. Érvényes tehát a következő:

Tétel: *Tegyük fel, hogy egy pusztán két kimenetelű döntéseket használó program legfeljebb k ilyen döntés alapján rendezi n elem bármelyik bemeneti sorrendjét. Ekkor $k \geq \log_2(n!)$ teljesül.* □

Példa: Az előző gondolatmenet illusztrálásaként vegyük a buborék-rendezést az $n = 3$ esetben. A ciklusok kifejtése után a program így fest:

$$
\begin{aligned}
&\textbf{if } A[2] < A[1] \textbf{ then } A[2] \leftrightarrow A[1];\\
&\textbf{if } A[3] < A[2] \textbf{ then } A[3] \leftrightarrow A[2];\\
&\textbf{if } A[2] < A[1] \textbf{ then } A[2] \leftrightarrow A[1].
\end{aligned}
$$

Itt az $A[i] \leftrightarrow A[j]$ akció az $A[i]$ és $A[j]$ tartalmának cseréjét jelenti. Tegyük fel, hogy az 1,2,3 számok permutációit rendezzük. A következő kis táblázat megadja az egyes permutációkhoz tartozó kódszavakat. Látható, hogy hat különböző kódszót kaptunk.

Bemenet	kódszó
123	*nem, nem, nem*
132	*nem, igen, nem*
213	*igen, nem, nem*
231	*nem, igen, igen*
312	*igen, igen, nem*
321	*igen, igen, igen*

Az összehasonlítás alapú rendezések eleget tesznek az alsó becslésnél használt modell kikötésének. Ha az input sorozat (tömb, lista) elemei mind különbözők, akkor a $<$ reláció tesztelésével tényleg két kimenetelű döntéseket használunk. Egyenlőség nem fog előfordulni. A ciklusokkal kapcsolatos tesztek nem érdekesek. Az előző példához hasonlóan adott n-re a ciklusok kiküszöbölhetők. Érvényes ezért az alábbi:

Következmény: *Egy összehasonlítás alapú rendező módszer n elem rendezésekor legalább* $\log_2(n!)$ *összehasonlítást használ.* □

A beszúrásos rendezés jól megközelíti ezt a korlátot. Hátulütője viszont, hogy n^2-tel arányos számú mozgatást igényel, és így az összköltsége nagy lesz. Ezután egy olyan módszert ismertetünk, ami a mozgatások számában is állja a versenyt.

2.2.4. Összefésüléses rendezés

Két már rendezett sorozat (tömb, lista, stb.) tartalmának egy sorozatba való rendezése egyszerű feladatot jelent. Tegyük fel, hogy az $A[1:k]$ valamint a $B[1:l]$ tömbök rendezettek, és tartalmukat rendezetten tárolni akarjuk a $C[1:k+l]$ tömbben. A $C[1]$-be nyilván az $A[1]$ és $B[1]$ elemek közül a nem nagyobb kerül. Ha ez mondjuk $A[1]$, akkor az $A[2]$ és $B[1]$ egyike lesz $C[2]$. Általában, ha az $A[1:i]$ és $B[1:j]$ résztömböket már feldolgoztuk, tartalmuk nem csökkenően $C[1:i+j]$-ben van, akkor nyilván

$$C[i+j+1] := \min\{A[i+1], B[j+1]\}$$

a jó választás. Ezt az eljárást *összefésülésnek* nevezzük (jelölése: MERGE). Minden egyes összehasonlítás után egy elem a helyére kerül, kivéve az utolsót, amivel két elem helyét derítjük ki. A költség tehát $k+l-1$ összehasonlítás és $k+l$ mozgatás. Az összefésülés igen hatékony eljárás, hiszen a bemenő adatok hosszával arányos időben, sőt lényegében *egyetlen végigolvasással* elrendezi a két sorozatot. Hatékonysága miatt előszeretettel alkalmazzák a legváltozatosabb adatkezelő feladatokban.

Ilyen alkalmazásnak számít az *összefésüléses rendezés*. Az összefésüléses rendezés alapötlete, hogy először rendezzük külön-külön az $A[1:n]$ tömb első felét és második felét, majd ezek tartalmát fésüljük össze. Itt ismét az *oszd meg és uralkodj* elvvel találkozunk. Az $A[1:n]$ rendezésének a feladatát egy összefésülés árán visszavezetjük két feleakkora feladatra, az $A[1:\lceil n/2\rceil]$ és az $A[\lceil n/2\rceil+1:n]$ résztömbök rendezésére. E résztömböket természetesen ugyanezen ötlettel kívánjuk rendezni. Mindezek a következő rekurzív definíciót sugallják (az eljárás neve

MSORT, paramétere a rendezendő tömb). Ha $n > 1$, akkor

$$\text{MSORT}(A[1:n]) := \\ \text{MERGE}(\text{MSORT}(A[1:\lceil n/2\rceil]), \text{MSORT}(A[\lceil n/2\rceil+1:n])).$$

Hogy elvarrjuk a rekurzió alját, legyen MSORT($A[i,i]$) az üres utasítás. A módszer költségének elemzéséhez tegyük fel először, hogy $n = 2^k$. Legyen $T(n)$ a felhasznált összehasonlítások maximális száma n-elemű bemenetekre. Figyelembe véve az összefésülés költségét, a rekurzió szerint

$$T(n) \leq n - 1 + 2T(n/2),$$

ha $n > 1$. Ezt $n/2$-re alkalmazva és visszahelyettesítve

$$T(n) \leq n - 1 + 2(n/2 - 1 + 2T(n/4)) = n - 1 + 2(n/2 - 1) + 4T(n/4).$$

A felezést így folytatva végül

$$T(n) \leq n-1+2(n/2-1)+4(n/4-1)+\cdots+2^{k-1}(n/2^{k-1}-1) \leq n\lceil \log_2 n\rceil.$$

Figyelembe vettük itt, hogy $T(1) = 0$. Az utolsó becsléshez elhagytuk a -1 tagokat, és használtuk, hogy k a legnagyobb egész, melyre $n/2^{k-1} > 1$. (Természetesen itt n választása miatt $n/2^k = 1$ is teljesül, és ezért $\lceil \log_2 n\rceil$ helyett egyszerűen $\log_2 n = k$ is írható.)

Az elemzésnél tulajdonképpen a rekurzió mélysége szerinti *fázisokba* csoportosítva számláltuk össze az összefésüléseknél fellépő összehasonlításokat. Az első fázisban két $n/2$ méretű tömböt fésülünk össze, a következőben két-két $n/4$ méretűt, stb. Összesen k fázis van. Egy fázisban az összefésülések legfeljebb $2n$ mozgatással megvalósíthatók; ez összesen $2n\lceil \log_2 n\rceil$ mozgatás. A tárigény $2n$ cella: az összefésüléseket egy $C[1:n]$ segédtömbbe végezhetjük, majd az eredményt visszaírjuk A-ba.

Megjegyzés: Az összefésülés megoldható helyben is: tegyük fel, hogy az $A[1:2n]$ tömb $A[1:n]$ és $A[n+1:2n]$ részei rendezettek. Ekkor a két rész összefésülhető az A tömbbe $O(n)$ összehasonlítással úgy, hogy A-n kívül még legfeljebb csak konstans számú cellát (pl. változókat) használunk. A módszerek azonban elég bonyolultak és nem praktikusak.

Az összefésüléses rendezés költségére adott elemzésünknél feltettük, hogy n a 2 egy hatványa. Az általános esetben tetszőleges pozitív valós x-re legyen $T(x)$ az $\lceil x\rceil$ elem rendezésénél fellépő összehasonlítások maximális száma. Érvényes a

$$T(x) \leq \lceil x\rceil - 1 + 2T(x/2)$$

egyenlőtlenség, amiből a korábbihoz hasonlóan adódik $T(n) \leq n\lceil \log_2 n \rceil$ tetszőleges pozitív egész n-re.

A programozástechnikában járatos olvasó tudja, hogy a rekurzív eljárások többnyire kevésbé hatékonyak, mint az iterációt használók. (Az érem másik oldala, hogy rekurzióval gyakran egyszerűbb, könnyebben olvasható leírást kapunk.) Az összefésüléses rendezést is érdemesebb iteratív formában megírni. Ekkor az első lépésben az A-ban szomszédos párokat rendezzük. Utána a szomszédos párokból rendezett négyeseket alakítunk ki összefésüléssel, majd ezekből nyolcasokat, stb. Erre a megoldásra is érvényesek a rekurzív változat költségéről mondottak: $\lceil \log_2 n \rceil$ összefésülő fázisban legfeljebb $n\lceil \log_2 n \rceil$ összehasonlítást végzünk.

Az összefésüléses rendezés költsége – mindkét költségtényezőt figyelembe véve – $O(n \log n)$. Ugyanakkor az alsó becslésünk szerint $cn \log n$ összehasonlítás szükséges is. E két tényt összerakva megállapíthatjuk, hogy az összefésüléses rendezés *konstans szorzó erejéig optimális módszer.*

2.2.5. A kupac adatszerkezet és a kupacos rendezés

O brightening glance,
How can we know the dancer from the dance?
WILLIAM BUTLER YEATS

A következő rendezési módszerhez először egy *adatszerkezettel* fogunk megismerkedni. Adatszerkezeten bizonyos adat-fajták – *típusok* – és velük kapcsolatos funkciók, *műveletek* együttesét értjük. Bizonyosan mindenki számára ismerős a *lista* adatszerkezet. Egyetlen típusból áll, amit *elemnek* nevezünk. A műveletek elemek véges halmazainak a kezelésére szolgálnak. Ilyenek például az *első, megelőző, következő* elemeket megadó eljárások.

A *kupac* adatszerkezettel egy $(U, <)$ rendezett halmaz (a kulcsok univerzuma) egy S véges részhalmazát szeretnénk tárolni. Két művelet tartozik a szerkezethez. Az egyik a beszúrás: U egy elemének hozzávétele S-hez. A másik az S halmaz $<$ szerinti minimális elemének a törlése. Legyen ezen műveletek neve BESZÚR és MINTÖR. A kupac adatszerkezet tehát egy $(U, <)$ rendezett típus e két művelettel. Mire jó egy ilyen eljáráspár? A számos fontos alkalmazás közül itt hármat említünk.

1. Bizonyos (komolyabb) operációs rendszerekben a beérkező munkákhoz (jobokhoz) prioritások rendelhetők. A rendszer mindig a legkisebb prioritási értékű munkát indítja el először. A még el nem kezdett munkákat egy kupacban tárolhatjuk. Az operációs rendszerhez érkező munkákat BESZÚR műveletekkel tesszük a kupacba. A következő indítandó munkát egy MINTÖR adja; ezzel a munka egyben

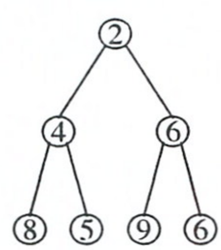
ki is kerül a kupacból. Ebből az alkalmazásból ered a kupac régebbi terminológia szerinti neve: *prioritási sor* (priority queue).
2. Több rendezett adatállomány összefésülése esetén célszerű lehet az egyes bemenő állományok legkisebb még feldolgozatlan elemeit egy kupacban tartani. A következő kiírandó elemet egy MINTÖR adja.
3. Egy gyors és bűbájos rendező eljárás, *a kupacos rendezés*, amit itt bemutatunk.

A kupac adatszerkezet egy hatékony implementációja: a bináris kupac
Emlékeztetünk a *bináris fa* fogalmára. Ez egy olyan fa, amelynek csúcsai szinteken helyezkednek el. A legfelső szinten egyetlen csúcs van, ezt *gyökérnek* nevezzük. Egy tetszőleges x csúcsból legfeljebb két él indul ki. Az élekhez irány tartozik (*bal, jobb*). Ezek eggyel alacsonyabb szinten levő csúcsokhoz vezetnek. A balra menő él végpontja az x *bal fia*, a jobbra menő él végpontja az x *jobb fia*. Egy a gyökértől különböző csúcsba pontosan egy él megy. Ennek megfelelően a gyökértől különböző csúcsoknak egyértelműen meghatározott *apjuk* van. Azokat a csúcsokat, amelyeknek nincs fiuk, *leveleknek* nevezzük. A nem-levél csúcsokat szokás *belső csúcsoknak* is nevezni.

Bizonyos szép, telt bináris fák jól ábrázolhatók tömbben, pontosabban egy tömb tekinthető bináris fának a következők szerint: a fa csúcsai az $A[1:n]$ tömb elemei. Az $A[i]$ csúcs bal fia $A[2i]$, a jobb fia pedig $A[2i+1]$. Ebből adódóan ha $j > 1$, akkor az $A[j]$ csúcs apja $A[\lfloor j/2 \rfloor]$ lesz.

Az ilyen alakban ábrázolható bináris fákat *teljesnek* nevezzük. A teljes fák levelei legfeljebb 2 szinten, a legalsón és esetleg a felette levőn helyezkednek el, és legfeljebb egy kivétellel minden belső csúcsuknak 2 fia van.

A kupac elemeit egy teljes bináris fában tároljuk. A fa csúcsaiban vannak a tárolni kívánt S halmaz elemei Egy csúcsban egy elem van; megeshet viszont, hogy egy $s \in S$ elem több csúcsban is előfordul. A fára teljesül a *kupac-tulajdonság*: egy tetszőleges csúcs eleme nem lehet nagyobb a fiaiban levő elemeknél. A következő példában az elemek természetes számok (a szokásos rendezéssel). A tárolt S halmaz – a többszörös elem jelenlétét is feltüntetve – $\{2, 4, 5, 6, 6, 8, 9\}$.



Ha a kupacnak (multiplicitásokkal számolva) n eleme van, akkor a keretül

szolgáló teljes bináris fa egy n elemű tömbben tárolható. Az előző példának megfelelő $A[1:7]$ tömb így fest:

2	4	6	8	5	9	6

Nézzük meg ezután, hogy miként építhető kupac n elemből. Először az elemeket beletesszük az $A[1:n]$ tömbbe, amit most teljes bináris fának is tekintünk. A sorrend egyelőre tetszőleges; mondjuk abban a sorrendben, ahogy olvassuk őket. Ezután gondoskodni kell a kupac-tulajdonságról.

Az eljárás leírásához hasznosak lesznek a következő jelölések: legyen f egy részfa gyökere, jelölje a az itt tárolt elemet, az f fiai legyenek f_1, f_2, az elemek itt a_1, illetve a_2.

Tegyük fel, hogy a kupaccá alakítás útján már tartunk valahol: az f_1 és f_2 gyökerű részfák már kupacok. Szeretnénk innen elérni, hogy az f gyökerű részfára is teljesüljön a kupac-tulajdonság. Ezt hivatott biztosítani a *felszivárog* eljárás.

felszivárog(f)
{ Ha $\min\{a_1, a_2\} < a$, akkor $\min\{a_1, a_2\}$ és a helyet cserél.
Ha az a elem a_i-vel cserélt helyet, akkor *felszivárog*(f_i). }

A feltételek szerint a részfában a kupac-tulajdonság csak az f csúcsnál sérülhet. Ha ez a helyzet, akkor az elemcsere után a gyökérrel és a másik részfával már nem lehet baj. Elég az f_i gyökerű részfát kupaccá tenni. Erre hivatott a *felszivárog*(f_i) hívás, aminek szintén teljesülnek a feltételei. Valójában az történik, hogy az a elem addig megy lefelé a fában, amíg sérti a kupac-tulajdonságot. Tehát ha előbb nem, egy levélben bizonyosan nyugvópontra jut. Egy lefelé lépés költsége 2 összehasonlítás és egy csere. Ha a részfának l szintje van, akkor ez legfeljebb $l-1$ csere és $2(l-1)$ összehasonlítás.

Legyen most f egy teljes bináris fa (gyökere), a fa csúcsaiban S elemeivel. A következő eljárás az egész fát kupaccá teszi.

kupacépítés(f)
{ Az f fa v csúcsaira lentről felfelé, jobbról balra *felszivárog*(v). }

Vegyük észre, hogy a *felszivárog* hívásainál teljesülnek a hívási feltételek; amikor a *felszivárog*(v) hívás sorra kerül, akkor v fiait már rendbetettük. A hívási sorrend az $A[1:n]$ tömbön nézve azt jelenti, hogy $A[n]$ felől haladunk $A[1]$ felé.
A *kupacépítés* eljárás költsége: tegyük fel, hogy a fának n eleme és l szintje van. (A gyökér az első szint.) Ekkor nyilván $n \leq 2^l - 1$. A fa teljessége miatt azt is tudjuk, hogy $2^{l-1} \leq n$, amiből $l \leq \log_2 n + 1$.

Az i-edik szinten levő csúcsok száma nem több, mint 2^{i-1}. Az i. szinten levő v csúcsra *felszivárog*(v) költsége legfeljebb $l-i$ csere és legfeljebb $2(l-i)$ összehasonlítás. A cserék száma ezért összesen legfeljebb $\sum_{i=1}^{l}(l-i)2^{i-1}$. A $j=l-i$ (azaz $i=l-j$) helyettesítés után a cserék száma így becsülhető:

$$\sum_{j=0}^{l-1} j2^{l-j-1} = 2^{l-1}\sum_{j=0}^{l-1} j/2^j < 2^l \leq 2n.$$

Csak az utolsó előtti egyenlőtlenség érdemel némi indoklást. A $j/2^j$ alakú tagok táblázatba foglalhatók, ahol egy tagnak egy sor felel meg:

$$\begin{array}{llll}
1/2 & & & \\
1/4 & 1/4 & & \\
1/8 & 1/8 & 1/8 & \\
\vdots & & & \\
\frac{1}{2^{l-1}} & \frac{1}{2^{l-1}} & \frac{1}{2^{l-1}} & \cdots \quad \frac{1}{2^{l-1}}
\end{array}$$

A táblázat oszlopainak összegei kisebbek mint 1, $1/2, \ldots, 1/2^{l-1}$, ezek összege pedig kisebb mint 2. A szükséges cserék száma tehát legfeljebb $2n$. Az összehasonlítások száma az előbbi összeg kétszeresével becsülhető; így nem lehet több, mint $4n$. Összegezésül elmondhatjuk, hogy a *kupacépítés* eljárás lineáris költséggel megvalósítható.

Nézzük ezután a kupacra jellemző alapműveleteket:
A MINTÖR megvalósítása: a törlendő elem a kupac-tulajdonság miatt a fa f gyökerében van. Ezt kivesszük innen, majd f-be tesszük a fa utolsó szintjén a jobb szélső csúcsban levő elemet; az utóbbi csúcsot töröljük, majd *felszivárog* (f) következik. Az összköltség $O(l) = O(\log n)$. Az $A[1:n]$ tömbön szemlélve mindezt: a hívás az $A[1]$ elemet adja vissza, $A[n]$ kerül $A[1]$-be, erre meghívjuk a *felszivárog* eljárást, majd a tömb méretét csökkentjük. Az új kupac $A[1:n-1]$-ben lesz.
BESZÚR(s) megvalósítása: új levelet adunk a fához (ügyelve a teljességre). Ebbe a levélbe tesszük az s elemet. Ezután s-et mozgatjuk felfelé, amíg az őt tartalmazó csúcs apjában levő s' elemre igaz, hogy $s < s'$. Az összköltség $O(l) = O(\log n)$. Mit jelent ez tömbös ábrázolásban? Az $A[1:n]$ tömbhöz új elemet veszünk. Az $A[n+1]$-be tesszük az s elemet, ami innen kezdi meg felfelé vándorlását.

A kupac adatszerkezet itt leírt implementációját, a *bináris kupac*ot szokás egyszerűen csak kupacnak nevezni. Az irodalomban gyakran előfordul, hogy a terminológia nem tesz különbséget a szerkezet és annak valamelyik megvalósítása között.

Bizonyos alkalmazásokban hasznos a FOGYASZT eljárás. Ennek célja egy a kupacban levő elem „kulcsának csökkentése"; pontosabban fogalmazva kisebbel való helyettesítése. Két bemenő paramétere van; az egyik a szerkezet egy s eleme, a másik az U egy s-nél kisebb s' eleme. Az s-ről feltesszük, hogy ismerjük a kupacbeli helyét, tehát nem kell megkeresni. FOGYASZT(s, s') az s helyére az $s' < s$ elemet teszi, majd helyreállítja a kupac-tulajdonságot. Utóbbi a beszúrásnál látott módon az s' felfelé való mozgatásával érhető el. A művelet költsége ezért $O(l) = O(\log n)$.

A kupacos rendezés

A kupacos rendezés módszerét J. W. J. Williams és R. W. Floyd javasolták 1964-ben. Bemenete a rendezendő sorozatot tartalmazó $A[1 : n]$ tömb. Először kupacot építünk, utána n darab MINTÖR adja nem csökkenő sorrendben az elemeket. Az összköltség az előzőek alapján $O(n) + O(n \log n) = O(n \log n)$. A kupacos rendezés elméletileg a leggyorsabb ismert rendező módszer. Ez úgy értendő, hogy az összesített költségére ismert felső becslések a legjobbak a rendező eljárások korlátai közül. Gondos, pontos implementáció és becslés mellett a korlátok $2n\lfloor \log_2 n \rfloor + 3n$ (összehasonlítások száma) és $n\lfloor \log_2 n \rfloor + 2,5n$ (cserék száma).

A kupac egy általánosabb implementációja: d-kupacok

A kupac adatszerkezet egy olyan implementációját ismertük meg, amelyben az elemeket tartalmazó alapstruktúra egy tömbben ábrázolt bináris fa. Pontosabban szólva az $A[1 : n]$ tömb elemeit egy bináris fa csúcsainak tekintettük. Ez a szemlélet könnyen általánosítható. Legyen $d > 1$ egy egész. Az A tömbön egy olyan gyökeres fa-struktúrát definiálunk, amelyben egy csúcsnak legfeljebb d fia van. A fa gyökere legyen itt is $A[1]$. Az $A[i]$ csúcs fiai legyenek az $A[d(i-1)+2], A[d(i-1)+3], \ldots A[di+1]$ elemek, illetve ezek közül azok, amelyek indexe nem nagyobb, mint n. Ennek megfelelően $i > 1$-re az $A[i]$ csúcs apjának indexe $\lceil (i-1)/d \rceil$. Nevezzük az így nyert fát *teljes d-fának*. Egyszerűen meggyőződhetünk róla, hogy a $d = 2$ esetben éppen a teljes bináris fákat kapjuk. Egy teljes d-fa levelei legfeljebb 2 szinten helyezkednek el, és legfeljebb egy kivétellel minden nem-levél csúcsának éppen d fia van. Ennek folyományaként ha a fa l szintből áll, akkor az első $l-1$ szinten $1 + d + d^2 + \cdots + d^{l-1} = (d^l - 1)/(d-1)$ csúcs van. Innen $d^l/(d-1) \leq n$, és d alapú logaritmusra térve $l \leq \log_d n + 1$.

A kupac adatszerkezet *d-kupac* elnevezésű implementációjában az S elemeit egy teljes d-fa csúcsaiban helyezzük el úgy, hogy teljesüljön a kupac-tulajdonság: egy csúcs eleme nem lehet nagyobb a fiaiban levő elemeknél. A korábbi implementációra – amit immár 2-kupacnak is hívhatunk – megismert algoritmusok kézenfekvően általánosíthatók d-kupacokra.

Feladat: Mutassuk meg, hogy egy n elemű d-kupac esetén az alapeljárások megvalósíthatók az alábbi táblázatban foglalt költségkorláton belül. Az f egy olyan csúcsot jelöl, amelynek a részfája k szintből áll:

Eljárás	Műveletigény
felszivárog(f)	$O(dk)$
kupacépítés	$O(n)$
MINTÖR	$O(d \log_d n)$
BESZÚR	$O(\log_d n)$
FOGYASZT	$O(\log_d n)$

(A módszerek a $d = 2$ esetre látottak kézenfekvő általánosításai. A *kupacépítés* elemzéséhez hasznos, hogy a fa i-edik szintjén legfeljebb d^{i-1} csúcs van, és $\sum_{i \geq 0} i/d^i \leq \sum_{i \geq 0} i/2^i \leq 2$.)

2.2.6. Gyorsrendezés

A módszert C. A. R. Hoare találta 1960-ban. Az alapötlet ismét egy szép példa az *oszd meg és uralkodj* elv alkalmazására. A rendezendő $A[1:n]$ tömb egy véletlen s elemét vesszük. Ezután a tömb elejébe mozgatjuk az s-nél kisebb elemeket, a végébe az s-nél nagyobbakat. A kettő között helyezkednek el az s előfordulásai. Nevezzük PARTÍCIÓnak az eljárást, ami ezt megvalósítja. A PARTÍCIÓ(s) tehát felbontja három részre az A tömböt. Az $A[1:k]$ résztömb elemei s-nél kisebbek, $A[k+1:l]$ minden eleme s, végül $A[l+1:n]$ elemei s-nél nagyobbak. A PARTÍCIÓ(s) hívás utáni helyzet így szemléltethető:

s-nél kisebb elemek	s	$\ldots$	s	*s-nél nagyobb elemek*

Világos, hogy eztán már elegendő az első és utolsó résztömböt rendezni. A GYORSREND eljárás vázlata következik.

GYORSREND($A[1:n]$)
1. Válasszunk egy véletlen s elemet az A tömbből.
2. PARTÍCIÓ(s); az eredmény legyen az $A[1:k]$, $A[k+1:l]$, $A[l+1:n]$ felbontás.
3. GYORSREND($A[1:k]$); GYORSREND($A[l+1:n]$).

A GYORSREND-algoritmus lelke a partícionáló eljárás. Érdemes ezért jobban szemügyre venni. Tegyük fel, hogy az $A[1:n]$ tömbbel és az s elemmel dolgozunk. Az i és j változók értéke kezdetben 1, illetve n. Először i-t növeljük, amíg $A[i] < s$ teljesül. Utána j-t csökkentjük, amíg $A[j] \geq s$.

	$i \rightarrow$ $\quad$ $\leftarrow j$	
s-nél kisebb elemek		*s-nél nem kisebb elemek*

Ha mindkettő megáll (nem lehet továbblépni), és $i < j$, akkor kicseréljük $A[i]$ és $A[j]$ tartalmát. Ezután $i := i + 1$ és $j := j - 1$. Ha a két mutató összeér (már nem teljesül $i < j$), akkor s előfordulásait a felső rész elejére mozgatjuk. Innen világos, hogy PARTÍCIÓ(s) költsége $O(n)$.

A GYORSREND eljárás futási ideje nagymértékben függ attól, hogy a partícionáló elem mekkora méretű részekre (mennyire egyenletesen) vágja szét a tömböt. (Gondoljuk meg, mi történik, ha mindig a rendezés szerinti legkisebb elemet használjuk.) Itt van szerepe a véletlen választásnak. Ekkor nagy az esélye, hogy s eléggé egyenletes vágást biztosít.

A gyorsrendezés átlagos költsége

Itt becslést adunk a gyorsrendezés során fellépő összehasonlítások átlagos számára. Az egyszerűség kedvéért feltesszük, hogy a rendezendő $A[1 : n]$ tömb elemei az $1, 2, \ldots, n$ számok. Jelölje $C(n)$ az A rendezéséhez felhasznált összehasonlítások átlagos számát, ahol az átlagot az $n!$ lehetséges bemeneti sorrendre vesszük. Nyilván igaz, hogy $C(0) = C(1) = 0$. Legyen $C(n, j)$ az összehasonlításokban mért átlagos költség abban az esetben, amikor a legelőször választott partícionáló elem éppen j. A módszer szerkezetéből látjuk, hogy

$$C(n, j) = n - 1 + C(j - 1) + C(n - j).$$

Az első tag a partícionálás összehasonlításaiból adódik, a következő kettő pedig az alsó, illetve a felső rész rendezéséhez használt összehasonlítások átlagos száma. A feltételeink szerint mindegyik j ugyanakkora eséllyel lesz az első partícionáló elem. Érvényes tehát a következő:

$$C(n) = \frac{1}{n}\left(C(n, 1) + C(n, 2) + \cdots + C(n, n - 1) + C(n, n)\right).$$

Innen az előző egyenleteket használva kiküszöbölhetjük a $C(n, j)$ mennyiségeket:

$$(+) \qquad C(n) = n - 1 + \frac{2}{n}\left(C(1) + C(2) + \cdots + C(n - 1)\right),$$

amiből n-nel való szorzás után

$$nC(n) = n(n - 1) + 2\left(C(1) + C(2) + \cdots + C(n - 1)\right).$$

A kapott kifejezés hátulütője, hogy túl sok $C(i)$ alakú tag szerepel benne. Ezen segíthetünk, ha n helyett $n - 1$-re is felírjuk a formulát, majd ezt kivonjuk $(+)$-ból:

$$nC(n) - (n - 1)C(n - 1) = 2(n - 1) + 2C(n - 1).$$

Innen átrendezés és $n(n+1)$-gyel való osztás után a következő egyenlőséget nyerjük:

$$\frac{C(n)}{n+1} = \frac{2(n-1)}{n(n+1)} + \frac{C(n-1)}{n},$$

ami elég egyszerű szerkezetű rekurziót ad a $C(n)/(n+1)$ mennyiségekre. Mivel csak felső korlátot akarunk, további egyszerűsítést érhetünk el a jobboldal első tagjának kis növelésével. A számlálóban $n-1$ helyett $n+1$-et írva

$$\frac{C(n)}{n+1} < \frac{2}{n} + \frac{C(n-1)}{n}.$$

Ezt ismételten önmagába helyettesítve végül azt nyerjük, hogy

$$\frac{C(n)}{n+1} < \frac{2}{n} + \frac{2}{n-1} + \cdots + \frac{2}{3} + \frac{2}{2} + \frac{2}{1}.$$

A jobboldal a nevezetes $H_n = 1 + 1/2 + 1/3 + \cdots + 1/n$ *harmonikus szám* kétszerese. A H_n számok nagyságrendjéről ismert, hogy $H_n = \ln n + \gamma + o(1)$, ahol $\gamma = 0.55721\ldots$ az Euler-állandó. Mindezeket összeolvasva azt kapjuk, hogy

$$C(n) < 2n\ln n + O(n) \approx 1,39n\log_2 n + O(n).$$

Itt használtuk, hogy $\ln n = (\ln 2)\log_2 n$ és $\ln 2 \approx 0,69$.

A módszer tehát átlagosan csak mintegy 39 százalékkal használ több összehasonlítást annál, amit az alsó korlát megkövetel. Valójában a gyorsrendezés átlagos értelemben az egyik leggyorsabb ismert módszer annak ellenére, hogy a legrosszabb esetben akár cn^2 is lehet a költsége. Gyakorlati viselkedését tekintve a *legjobbnak tartott* összehasonlítás alapú módszer memóriában levő állományok (lista, tömb) rendezésére.

2.2.7. A k-adik elem kiválasztása

Tegyük fel, hogy adott az $(U, <)$ rendezett típusból való elemek egy $u_1, u_2, \ldots, u_n$ sorozata, és egy k egész, $(1 \leq k \leq n)$. A feladat az, hogy határozzuk meg a sorozatnak a $<$ rendezés szerinti k-adik legnagyobb elemét.

A $k = n$ eset a maximális elem(ek egyikének) kiválasztását jelenti. Már a buborék-rendezés kapcsán láttuk, hogy ez megtehető $n-1$ összehasonlítással. Ennél kevesebb általában nem elég. Ahhoz ugyanis, hogy a maximálistól különböző u elemről tényleg biztosan tudjuk, hogy nem maximális, szükség van egy olyan összehasonlításra, amelynél u kisebbnek bizonyul. A maximumtól különböző $n-1$ elem mindegyike legalább egyszer alulmaradt. Mivel egy ilyen mérkőzésnek legfeljebb egy vesztese van, a mérkőzések száma legalább $n-1$. Hasonló mondható a $k = 1$ esetről, a minimális elem kiválasztásáról.

A második legnagyobb elemet $n + \lceil \log_2 n \rceil - 2$ összehasonlítással azonosíthatjuk. Egy lehetséges módszer a sportból jól ismert kieséses verseny. Állítsuk az elemeket párokba, hasonlítsuk össze a párok tagjait, majd a győztesekkel járjunk el ugyanígy. Végül $\lceil \log_2 n \rceil$ fordulóban összesen $n - 1$ mérkőzést játszva kiderül, hogy ki a győztes. A második helyezett nyilván azok között van, akik csak a bajnok ellen vesztettek. Ilyen jelöltekből legfeljebb $\lceil \log_2 n \rceil$ lehet; $\lceil \log_2 n \rceil - 1$ mérkőzéssel eldönthető, hogy közülük ki a legjobb. Az is igaz, hogy $n + \lceil \log_2 n \rceil - 3$ összehasonlítás nem mindig elég. Ezt nem részletezzük.

Általában elmondhatjuk, hogy $O(n \log n)$ összehasonlítással tetszőleges k-ra célt érünk. Ekkora költséggel rendezhető az u_i sorozat, ami után a k-adik elem könnyen meglelhető.

Ha $k \leq n/\log n$, akkor lineáris számú, azaz $O(n)$ összehasonlítás elég: építsünk kupacot az u_i elemekből, majd hajtsunk végre k egymás utáni MINTÖR-t. A költség $O(n) + kO(\log n) = O(n) + O((n/\log n)\log n) = O(n)$. Ugyanez érvényes, ha $k \geq n - n/\log n$.

A kiválasztás feladata általában is megoldható lineáris költséggel. Az ilyen módszerek azonban meglehetősen bonyolultak és csak nagy n értékekre tudnak versenyezni az egyszerűbbekkel, mint amilyen a teljes rendezésre építő eljárás. Mindezek ellenére érdemes megismerkedni e módszerek boszorkányos redukciós ötletével.

Tegyük fel tehát, hogy a k-adik elemet akarjuk kiválasztani. Jelölje $T(n)$ az ismertetésre kerülő módszer összehasonlításainak maximális számát n elem és tetszőleges k mellett. Az egyszerűség kedvéért feltesszük még, hogy az u_i elemek mind különbözők. Olyan elemet igyekszünk találni – nevezzük ezt *polgárnak* –, ami mindkét szélsőségtől távol van abban az értelemben, hogy legalább $n/4$ elem kisebb nála, és legalább $n/4$ elem nagyobb nála. Ha van egy ilyen elemünk, akkor $n - 1$ összehasonlítással megkaphatjuk a nála kisebb elemek S_1 és a nála nagyobb elemek S_2 halmazát. Ha $k < |S_1|$, akkor ezután elég S_1 k-adik elemét megtalálni, ha pedig $k > |S_1| + 1$, akkor S_2-nek a $k - |S_1| - 1$-edik eleme lesz az új célpont. Mindkét esetben kevesebb, mint $3n/4$ elem közül kell választani. Egy polgár segítségével tehát drámai módon csökkenthető a feladat mérete. A gondolatmenetből a $T(n)$ függvényre az alábbi egyenlőtlenség következik:

$$T(n) \leq n - 1 + \textit{ a polgár költsége } + T(3n/4).$$

Világos innen, hogy akkor kapunk erős korlátot $T(n)$-re, ha gyorsan tudunk polgárt találni. Ezt pedig a következőképpen tesszük:

1. Az n elemet ötös csoportokba soroljuk, elhagyva a végén keletkező 0-4-elemű csoportocskát. Összesen $\lfloor n/5 \rfloor$ ilyen csoportunk lesz.

2. Minden ötös csoportnak meghatározzuk a rendezés szerinti középső elemét; ez csoportonként legfeljebb 6 összehasonlítás, összesen nem több, mint $6\lfloor n/5 \rfloor$.
3. Az előbb meghatározott $\lfloor n/5 \rfloor$ elem közül kiválasztjuk a középsőt. Legyen ez a polgár. Ez az $\lfloor \frac{n+5}{10} \rfloor$-edik elemet jelenti. A választást az éppen tárgyalt kiválasztó algoritmussal végezzük. Ha úgy tetszik, itt egy *rekurzív hívás* történt.

A polgár nem kisebb az n elem közül $3\lfloor \frac{n+5}{10} \rfloor$-nél. Ugyanis nem kisebb, mint $\lfloor \frac{n+5}{10} \rfloor$ középső elem, amelyek nagyobbak a csoportjukban még további két-két elemnél. Egyszerű számolás mutatja, hogy ha $n \geq 25$, akkor $3\lfloor \frac{n+5}{10} \rfloor > n/4$. Eszerint a választott elemünk nagyobb legalább $n/4$ elemnél. Hasonló érveléssel kapjuk, hogy ha $n \geq 25$, akkor a polgár kisebb legalább $n/4$ elemnél. Tényleg rászolgál tehát a polgár névre. Ha $n \leq 24$, akkor a k-adik elemet közvetlenül (polgár választása nélkül) megkaphatjuk. Ez – összefésüléses rendezéssel – legfeljebb $n\lceil \log_2 n \rceil \leq 5n$ összehasonlítás.

Mibe kerül a polgárválasztás? A második lépés költsége legfeljebb $6n/5$ összehasonlítás, a harmadiké legfeljebb $T(n/5)$. A $T(n)$-re nyert egyenlőtlenség ezekkel az adatokkal így alakul:

$$T(n) \leq \begin{cases} 5n & \text{ha } n \leq 24 \\ 11n/5 + T(n/5) + T(3n/4) & \text{ha } n > 24. \end{cases}$$

Innen egyszerű indukcióval következik, hogy $T(n) \leq 44n$. Ez nyilván igaz, ha $n \leq 24$. Nagyobb n-ekre az indukciós feltevést használva

$$\begin{aligned} T(n) &\leq 11n/5 + T(n/5) + T(3n/4) \leq 11n/5 + 44n/5 + 44 \cdot 3n/4 = \\ &= 44n/20 + 4 \cdot 44n/20 + 15 \cdot 44n/20 = 20 \cdot 44n/20 = 44n. \end{aligned}$$

Pontosabb és egyszersmind bonyolultabb becsléssel a korlát $7n$-re levihető. Ezt nem részletezzük.

2.3. Kulcsmanipulációs rendezések

Az eddig tárgyalt módszerek igen keveset tételeztek fel az $(U, <)$ rendezett típusról (univerzumról), amihez a rendezendő elemek tartoznak. Kizárólag $s < s'$ alakú összehasonlítások eredményei alapján oldották meg a feladatot. Gyakran a $<$ reláción kívül sok mást is tudunk U-ról. Ismerhetjük például az elemei számát, vagy az elemek (kulcsok) belső szerkezetét. Ilyen ismeretekre építve az eddigieknél hatékonyabb rendezési módszereket adhatunk.

2.3.1. Ládarendezés (binsort)

Arról az eljárásról lesz itt szó, amit feltehetőleg Jim használt a Kincses Szigeten, amikor a sokféle érmét szortírozta. A módszer akkor hasznos, ha a lehetséges elemek U tartománya n-hez, a rendezendő elemek számához képest nem túl nagy. Tegyük fel, hogy az $A[1:n]$ tömböt szeretnénk rendezni, és tudjuk, hogy A elemei az m elemű U univerzumból kerülnek ki. Ekkor lefoglalhatunk egy U elemeivel indexelt B tömböt (U egy rendezett típus!). Ennek elemei - a ládák - U-beli elemek listái lesznek. Kezdetben minden lista (láda) üres. Az eljárás első fázisában végigolvassuk az A tömböt, és az $s = A[i]$ elemet a $B[s]$ lista végére fűzzük. A második fázisban az elejétől a végéig növő sorrendben végigmegyünk B-n, és a $B[i]$ listák tartalmát visszaírjuk A-ba.

Az eljárás költsége $O(n+m)$ elemi lépés. Létrehozzuk az üres listákból álló B tömböt. Erre $O(m)$ költséget számíthatunk. Az első fázis, A végigolvasása és az elemeinek listákba illesztése $O(n)$ elemi lépést jelent. Végül a második fázisban a listák olvasása és az elemek visszatevése A-ba további $O(m+n)$ elemi lépéssel megtehető.

Eszerint, ha az U kulcstartomány n-hez képest lineáris méretű (másként mondva: $m \leq cn$ teljesül egy c állandóval), akkor ládarendezéssel $O(n)$ költséggel, azaz lineáris időben tudunk rendezni. Ez kedvezőbb, mint amit összehasonlítás alapú módszerekkel elérhetünk. Javasoljuk az olvasónak: gondolja meg, miért nem érvényes itt az összehasonlító módszerekre adott alsó becslés.

Példa: Tegyük fel, hogy a rendezendő $A[1:7]$ tömb elemei 0 és 9 közötti egészek:

A:

5	3	1	5	6	9	6

Ekkor egy tízelemű $B[0:9]$ tömböt lesz célszerű használni. Az első fázis után B így képzelhető el:

B:

	1		3		5 5	6 6			9

A $B[0]$, $B[2]$, $B[4]$, $B[7]$, $B[8]$ listák üresek, hiszen A-ban nem fordulnak elő a 0, 2, 4, 7, 8 értékek. A $B[5]$ és $B[6]$ listák pedig kételeműek, mert az 5 és a 6 kétszer szerepelnek A-ban. Ebben az esetben valamivel egyszerűbben is eljárhattunk volna. A B elemei lista helyett lehettek volna egész típusúak. A $B[j]$-t ekkor számlálóként használjuk: $B[j]$ a j kulcs A-beli előfordulásainak a száma. Az első fázis utáni helyzet így nézne ki:

B:	0	1	0	1	0	2	2	0	0	1

A rendezése pusztán e számok ismeretében is elvégezhető. Listákra akkor van szükségünk, ha a rendezés alapját jelentő kulcsok mellett még más információk is vannak az $A[i]$ elemekben; például a kulcs csak egy mező valamely rekord típusú elemben. Rendezéskor a többi mezőt is mozgatni kell.

Feladat: A ládarendezés érdekes speciális esete, amikor $m = n$, más szóval a rendezendő elemek és a lehetséges kulcsok száma egyenlő. Mutassuk meg, hogy a rendezés megoldható $O(n)$ költséggel *helyben* is, vagyis az A tömb mellett még legfeljebb állandó számú további tárcella felhasználásával.

2.3.2. Radix rendezés

Tegyük fel, hogy a rendezendő kulcsok összetettek, több komponensből állnak, és a $<$ reláció a komponensek rendezéséből származó lexikografikus rendezés. Legyenek mondjuk az U elemei k hosszúságú $t_1 \dots t_k$ alakú szavak, ahol a t_i komponens az L_i rendezett típusból való. Tegyük fel még, hogy az L_i típusnak összesen s_i eleme van. Legyen $t_1 \dots t_k$ és $u_1 \dots u_k$ két kulcs U-ból. A lexikografikus (szótárszerű) rendezés definíciója szerint $t_1 \dots t_k > u_1 \dots u_k$ éppen akkor teljesül, ha $t_i > u_i$ a legkisebb i indexre, amelyre $t_i \neq u_i$.

Példa: Legyen $(U, <)$ a *huszadik századi dátumok* összessége az időrendnek megfelelő rendezéssel. U természetes módon tekinthető három komponensből álló összetett típusnak. Az első az évszámokból álló típus:

$$L_1 = \{1900, 1901, \dots, 1999\}, \quad s_1 = 100.$$

A második a hónapokat magába foglaló típus:

$$L_2 = \{\textit{január, február}, \dots, \textit{december}\}, \quad s_2 = 12.$$

A harmadik összetevőt a hónapok napjai adják:

$$L_3 = \{1, 2, \dots, 31\}, \quad s_3 = 31.$$

A dátumok rendezése éppen az L_i típusokból származó lexikografikus rendezés lesz.

A radix rendezés ilyen értelemben összetett kulcsok sorozatainak a rendezésére szolgál. A receptje lefegyverzően egyszerű: először rendezzük a sorozatot az utolsó, a k-adik komponensek szerint ládarendezéssel. Utána az eredményül kapott sorozatot ládarendezzük a $k-1$-edik komponens szerint, és így tovább, végül

az első komponens szerint. Arra kell ügyelni, hogy az elemeket mindig a ládájukat jelentő lista *végére* rakjuk. Más szóval, ha a j-edik ládarendezés előtt az X kulcs előbb van a sorozatban, mint az Y kulcs, és a kulcsok $k-j+1$-edik komponensei egyenlők, akkor X megelőzi Y-t a j-edik menet után is.

Az eljárás helyességét elég a $k = 2$ esetben vizsgálni; az érvelés könnyen általánosítható. Legyen $X = x_1x_2$ és $Y = y_1y_2$ két kulcs, és tegyük fel, hogy $X < Y$. Azt kell igazolni, hogy a végső sorozatban X előbb áll, mint Y. Ezt az utolsó, a második rendezés biztosítja, ha $x_1 < y_1$. Ha $x_1 = y_1$, akkor $X < Y$ miatt szükségképpen $x_2 < y_2$. Ekkor az első menet után X megelőzi Y-t. A kiegészítő szabályunk szerint ezen a viszonyon a második menet sem változtat; X a végső sorrendben is Y előtt lesz.

Példa: Nézzük a módszer működését a következő – dátum típusú – $A[1:5]$ tömbre mint bemenetre:

1969. jan. 18.	1969. jan. 1.	1955. dec. 18.	1955. jan. 18.	1918. dec. 18.

Itt $k = 3$. Az első fázisban a napok, utána a hónapok, végül az évek szerint rendezünk. Az egyes menetek utáni helyzeteket mutatja az alábbi ábra.

1969. jan. 1.	1969. jan. 18.	1955. dec. 18.	1955. jan. 18.	1918. dec. 18.

1969. jan. 1.	1969. jan. 18.	1955. jan. 18.	1955. dec. 18.	1918. dec. 18.

1918. dec. 18.	1955. jan. 18.	1955. dec. 18.	1969. jan. 1.	1969. jan. 18.

A radix rendezés költsége a k ládarendezés költségének az összege. A ládarendezésről mondottak szerint ez $O(kn + \sum_{i=1}^{k} s_i)$ elemi műveletet jelent. Ez bizonyos esetekben ismét jobbat adhat az összehasonlító módszerek idejénél. Nézzünk néhány példát.

Példák:
1. Tegyük fel, hogy k = állandó és $s_i \leq cn$ ($c > 0$ egy konstans). Ekkor a költség $O(kn + \sum_{i=1}^{k} cn) = O(k(c+1)n) = O(n)$. Ilyen helyzet például, amikor az $[1, n^k - 1]$ intervallumból való egészeket akarunk rendezni. Ezeket az egészeket k-jegyű számoknak vehetjük az n alapú számrendszerben. Az így felírt számok sorozatainak rendezésére a radix-módszer lineáris költségű megoldást ad.
2. $k = \log n$, $s_i = 2$. Ez a felállás, amikor $\log n$ hosszúságú bitsorozatokat rendezünk. Ekkor a radix rendezés költsége $O(n \log n + 2\log n) = O(n \log n)$.

2.4. A Batcher-féle páros-páratlan összefésülés

Itt egy hatékony párhuzamos rendező algoritmust ismertetünk. A módszer az összefésüléses rendezés egy változata. Az abban szereplő összefésülő eljárást cseréljük ki egy gyors párhuzamos megoldással.

Az összefésülő módszert a jelölés egyszerűsítésére törekedve sorozatokkal mondjuk el. Legyenek $\mathcal{A} = a_1 < \ldots < a_l$ és $\mathcal{B} = b_1 < \ldots < b_m$ az input sorozatok. Célunk ezek összefésülése egyetlen növekvő $\mathcal{C} = c_1 < \ldots < c_{l+m}$ sorozattá.

A feladatot két egymástól függetlenül és párhuzamosan végezhető részre bontjuk. Az első brigád összefésüli az $a_1 < a_3 < a_5 < \ldots$ és $b_2 < b_4 < b_6 < \ldots$ sorozatokat az $u_1 < u_2 < u_3 < \ldots$ sorozattá. A második brigád összefésüli a két megmaradó részt, az $a_2 < a_4 < a_6 < \ldots$ és $b_1 < b_3 < b_5 < \ldots$ sorozatokat a $v_1 < v_2 < v_3 < \ldots$ sorozatba. Ezzel látszólag ugyanott vagyunk, ahonnan elindultunk. Van két növekvő sorozatunk, az $\{u_i\}$ és a $\{v_j\}$, amiket össze kellene fésülni. A következő állítás azt mondja, hogy a látszat csal. Az utóbbi két sorozat párhuzamosan *egyetlen lépésben* összefésülhető. A végeredményt jelentő sorozat c_j eleme egy összehasonlítással megkapható.

Állítás: *Érvényesek a* $c_{2i-1} = \min\{u_i, v_i\}$ *és* $c_{2i} = \max\{u_i, v_i\}$ *egyenlőségek* $(1 \leq i \leq (l+m)/2)$.

Bizonyítás: Először azt látjuk be, hogy tetszőleges $1 \leq k \leq (l+m)/2$ egészre a $\mathcal{C}_k := c_1, \ldots, c_{2k}$ részsorozatban ugyanannyi u_i van, mint v_j. Ezt úgy is mondhatjuk, hogy $\{c_1, \ldots, c_{2k}\} = \{u_1, \ldots, u_k\} \cup \{v_1, \ldots, v_k\}$. A $\mathcal{C}$ sorozat a növekvő $\mathcal{A}$ és $\mathcal{B}$ sorozatok összefésülése. Emiatt a $\mathcal{C}$ első $2k$ elemét az $\mathcal{A}$ sorozat első néhány eleme és a $\mathcal{B}$ sorozat első néhány eleme adja. Legyen mondjuk $\mathcal{C}_k = \{a_1, \ldots, a_s\} \cup \{b_1, \ldots, b_{2k-s}\}$. Ekkor a $\mathcal{C}_k$-ba eső u_i elemek közül $\lceil s/2 \rceil$ tartozik az $\mathcal{A}$, és $\lfloor \frac{2k-s}{2} \rfloor$ tartozik a $\mathcal{B}$ sorozatba. Ugyanígy a v_j elemek közül $\lfloor s/2 \rfloor$ tartozik az $\mathcal{A}$, és $\lceil \frac{2k-s}{2} \rceil$ tartozik a $\mathcal{B}$ sorozatba. A $\mathcal{C}_k$-ba kerülő u_i elemek száma $\lceil s/2 \rceil + \lfloor (2k-s)/2 \rfloor$, az ilyen v_j elemeké pedig $\lfloor s/2 \rfloor + \lceil (2k-s)/2 \rceil$. A kettő egyenlősége következik az egyszerű számolással igazolható

$$\lceil s/2 \rceil + \lfloor (2k-s)/2 \rfloor = \lfloor s/2 \rfloor + \lceil (2k-s)/2 \rceil$$

azonosságból.

Nézzük ezután az állítást. Az észrevétel szerint $\{c_1, \ldots, c_{2(i-1)}\} = \{u_1, \ldots, u_{i-1}\} \cup \{v_1, \ldots, v_{i-1}\}$ és $\{c_1, \ldots, c_{2i}\} = \{u_1, \ldots, u_i\} \cup \{v_1, \ldots, v_i\}$. Ezeket összevetve $\{c_{2i-1}, c_{2i}\} = \{u_i, v_i\}$, amiből az állítás $c_{2i-1} < c_{2i}$ miatt nyilvánvaló. □

A fentiekben vázolt páros-páratlan felbontással az összefésülés feladatát visszavezettük két párhuzamosan végezhető feleakkora méretű feladatra és ezen felül párhuzamosan állandó időben elvégezhető további munkára, a $\min\{u_i, v_i\}$ és $\max\{u_i, v_i\}$ elemek meghatározására. Ezzel egy rekurzív eljárást körvonalaztunk. A két brigád ugyanígy osztja tovább a problémát. Legyen az algoritmus neve PM.

Tegyük fel, hogy az $\mathcal{A}$ és $\mathcal{B}$ sorozatok összhossza n, és van $\lfloor n/2 \rfloor$ processzorunk. Valójában úgynevezett CE (compare-exchange) elemek is megteszik. Egy CE elem két kulcsot összehasonlít, és ha kell, felcseréli őket. A PM eljárás összehasonlításokban mért párhuzamos költségének maximumát adott n-re jelölje $T_p(n)$. A PM eljárás szerkezetéből rögvest adódik a $T_p(n) \leq T_p(\lceil n/2 \rceil)+1$ egyenlőtlenség. Az első tag a két félméretű részfeladat párhuzamos költségét, a második pedig az $\{u_i\}$ és $\{v_j\}$ sorozatok összefésülését könyveli. Ebből az összefésüléses rendezés elemzésénél alkalmazott visszahelyettesítős érveléssel adódik, hogy $T_p(n) \leq \lceil \log_2 n \rceil$.

A PM eljárást használhatjuk az összefésüléses rendezésben. A definíció így módosul:

$$\mathrm{MSORT}(A[1:n]) := \mathrm{PM}(\mathrm{MSORT}(A[1:\lceil n/2 \rceil]), \mathrm{MSORT}(A[\lceil n/2 \rceil + 1 : n])).$$

Jelölje $t_p(n)$ a módszer párhuzamos költségét n elemű bemenetre. A tömb alsó, illetve felső felének a rendezése párhuzamosan végezhető. Ezért $t_p(n) \leq t_p(\lceil n/2 \rceil) + \lceil \log_2 n \rceil$. A második tag a párhuzamos összefésülés ideje. Ezt ismét a visszahelyettesítő módszerrel fejthetjük ki:

$$t_p(n) \leq 1 + 2 + \cdots + (\lceil \log_2 n \rceil - 1) + \lceil \log_2 n \rceil = \frac{(\lceil \log_2 n \rceil + 1)(\lceil \log_2 n \rceil)}{2}.$$

A Batcher-rendező párhuzamos költsége eszerint $t_p(n) = O(\log^2 n)$.

2.5. Külső tárak tartalmának rendezése

Az eddig tárgyalt rendező eljárások kapcsán hallgatólagosan feltettük, hogy az adataink a számítások menete során végig egy nagy elérési sebességű *belső memóriában* vannak. A feltételezés ott bújt meg az eddigi elemzésekben, hogy nem szenteltünk túl nagy figyelmet az adatok mozgatásához szükséges időnek. Bizonyos esetekben csak az összehasonlításokat számoltuk. Máskor figyeltük ugyan a cserék és más mozgatások idejét, de ezeket az összehasonlításokéval egyező nagyságrendű tényezőként kezeltük. Ez a szemlélet jól működik a belső memóriás számítások esetén. Az így nyert becslések megfelelnek a tényleges számítások

időviszonyainak. *Külső tárakon* (mágneslemezek, dobok, szalagok) elhelyezkedő adatok kezelésére viszont a feltételezés általában nem alkalmazható.

Egy belső memóriában levő szó elérése és olvasása legfeljebb néhány mikroszekundumot jelent. A mágneslemezek elérési ideje viszont az ezredmásodperces nagyságrendben van, diszketteknél pedig a másodpercet is meghaladhatja. A külső tárak elérési ideje tehát több ezerszerese is lehet a belsőkének. Külső tárakon levő adatok kezelésénél ezért komoly figyelmet kell fordítanunk az adatelérés/mozgatás költségeire.

A külső tárak másik fontos vonása, hogy egy írás/olvasás során nem csak egy szót mozgatunk, hanem egy nagyobb területet. Használható modellt kapunk az ilyen tárak működésére, ha úgy képzeljük, hogy az állományok az elérhetőség szempontjából folytonosan egymás után következő *lapokon* helyezkednek el. Egy lap egyetlen I/O művelettel kezelhető (olvasható, írható) terület a táron. A szokásos lapméret $512 \cdot k$ byte valamely kicsi k-val. A folytonosság bizonyos esetekben tényleges fizikai folytonosságot jelent, máskor pedig magasabb szintű (pl. az operációs rendszer által biztosított) eljárások adnak módot a következő lap értelmezésére. Egy lapon általában több rekord is elfér. A lapok beolvasása, kiírása igen költséges a belső memóriában végzett elemi lépésekhez képest. Átlagos alkalmazásoknál az I/O műveletek időigénye a futási idő döntő része. Ezért a külső tárat jelentős mértékben használó programok esetén alapvető hatékonysági szempont a *lapelérések számának minimalizálása*. A korábban tárgyalt rendezési módszerek, ötletek általában nem adnak jó eredményt, ha a rendezendő adatok külső tárban vannak. Tulajdonképpen egyetlen kivétel az összefésüléses rendezés.

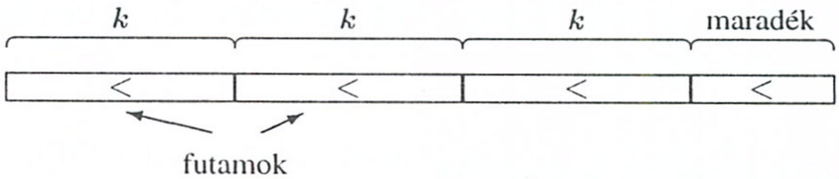
2.5.1. Összefésüléses rendezés külső tárakon

A bevezetőben említetteknek megfelelően az algoritmusok értékeléséhez egyetlen költségtényezőként a *lapelérések számát* használjuk. A módszerek, amiket ismertetünk, működnek mágnesszalagok esetén is, mert az állományokat lapjaik sorrendjében (szokásos szakkifejezéssel: *szekvenciálisan*) használjuk. Ez annyit tesz, hogy a bemeneti állományokat szekvenciálisan olvassuk, és így végezzük a kiírásokat is a kimeneti állományokba.

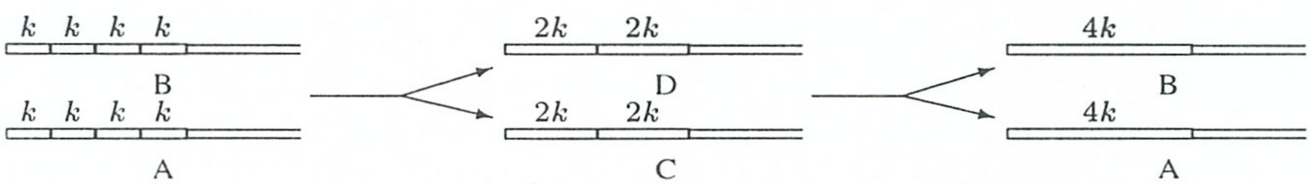
A célunk egy a háttértáron levő n lapból álló állomány rendezése valamilyen adott kulcs szerint (pl. személyek adatait tartalmazó rekordoké a személyi szám szerint).

Definíció: *Egy állomány k szomszédos lapjából álló rész k hosszú futam, ha a rekordok ebben a részben a kulcs szerint (nem csökkenően) rendezetten helyezkednek el.*

Először alkalmas kis k mellett (ha más nem, $k = 1$ mindig lehetséges) k hosszúságú kezdőfutamokat hozunk létre; ezeket egyenletesen elhelyezzük az A és B állományokba. Ez többnyire úgy történik, hogy egyszer végigolvassuk a bemenő állományt. A memóriában k lapnyi futamokat alakítunk ki; amint egy futamot befejeztünk, azt kiírjuk az A és B állományok egyikébe. Ennek a műveletnek a költsége $2n$ lapelérés. A futamok közül az utolsó esetleg k-nál kevesebb lapot tartalmaz.



Ezután az A és B állományok elejéről indulva sorban összefésüljük a futamokat – mindig egyet A-ból és egyet B-ből véve – $2k$ hosszúakká; ezeket egyenletesen elhelyezzük a C és D állományokba. Egy ilyen menet költsége $2n$ I/O művelet. Ezután C és D futamait fésüljük ugyanígy össze $4k$ hosszú futamokká A-ba és B-be, és így tovább.



Mint a kezdeti helyzetnél, később is előfordulhat, hogy a menetben utoljára kapott futam rövidebb a többinél. Ha az i-edik összefésülő menet után még egynél több futam van, akkor ezek közül az első hossza legalább 2^i. Ez pedig csak addig lehetséges, amíg $2^i < n$ teljesül. Ha a j-edik az utolsó menet, akkor a $j-1$. menet után még legalább két futam van, tehát $2^{j-1} < n$, amiből logaritmusra térve $j \leq \lceil \log_2 n \rceil$. A szükséges menetek száma nem több, mint $\lceil \log_2 n \rceil$. Az összes lapelérések száma ennek megfelelően legfeljebb $2n(\lceil \log_2 n \rceil + 1)$. Ezt az algoritmust, illetve változatait, finomításait használják leggyakrabban külső rendezésre.

Két gyorsítási lehetőség:

1. Érdemes minél nagyobb kezdő futamokat létrehozni, más szóval nagy k értékkel dolgozni. Az előbbi gondolatmenet pontosítható: ha k hosszúságú futamokkal kezdjük a fésülést, akkor $k2^{j-1} < n$ is teljesül, amiből a menetek számára $j \leq \lceil \log_2(n/k) \rceil$ adódik.

2. m-felé fésülés, ahol $m > 2$. Ekkor minden menetben m input és m output állományt használunk, és egyszerre m futamot fésülünk össze egyetlen futamba. Egy menet után a futamok mérete m-szereződik. A menetek száma legfeljebb $j = \lceil \log_m n \rceil = \lceil (\log_2 n)/\log_2 m \rceil$. Itt még további $2m$-szeres gyorsítás érhető el, ha van $2m$ párhuzamosan használható adatcsatornánk (minden állományhoz egy). Ekkor ugyanis egy I/O művelet ideje alatt olvashatunk egy-egy lapot az input-állományokból, és ugyanekkor írhatunk is egy-egy lapot a kimenő állományokba. Az m felé fésülésnél ügyelni kell arra, hogy m növekedtével a futamok összefésülésének költsége is nő. Egyre több idő kell, és egyre nagyobb területet használunk a belső memóriában.

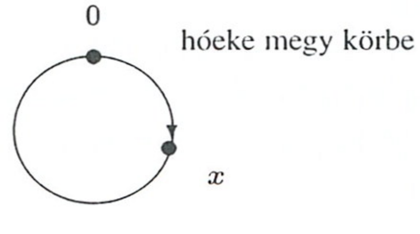
Kezdeti futamok létrehozása kupaccal

Itt egy érdekes algoritmust ismertetünk, amivel átlagos értelemben viszonylag hosszú kezdőfutamokat lehet építeni. A futamok hossza változhat, így előfordulhatnak rövid futamok is.

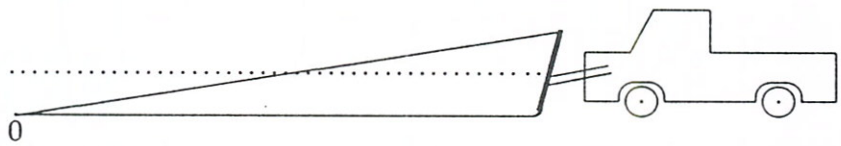
A belső memóriában lefoglalunk egy területet, amin K kulcs elfér. Mint látni fogjuk, K értékét érdemes minél nagyobbnak választani. A lefoglalt területen két – kulcsokat tartalmazó – kupacot üzemeltetünk. A két kupacban összesen legfeljebb K kulcsot tartunk. Az egyik lesz az aktív kupac a, a másik az alvó kupac A. Az aktív kupacban tartjuk az éppen kiírandó futam kulcsait, az alvóban pedig a következő futam kulcsait gyűjtjük. Kezdetben mindkettő üres. Először a-t feltöltjük a rendezendő állományból K kulccsal, A pedig üres.

Ezután amíg a nem üres, ismételjük a következő lépést. Az s =MINTÖR(a) kulcsú rekordot beírjuk az aktuális futamba. Ezután vesszük az input állomány következő rekordját. Legyen ennek a kulcsa t. Ha $t < s$, akkor a rekord már nyilván nem tehető az aktuális futam végére, ezért t-t A-ba szúrjuk be. Ha $t \geq s$, akkor a t kulcs az a-ba kerül, hiszen ekkor a rekord még elhelyezhető valahol később az aktuális futamban. Ha a kiürül, akkor a futam befejeződik. A két kupac szerepet cserél, A lesz aktív, és a megy aludni. Új futam kezdődik, amit A-ból töltünk az előzőek szerint.

Megmutatható, hogy – rekordokban mérve – az átlagos futamhossz $2K$ lesz, ha a kulcsok alkalmas értelemben egyenletesen érkeznek. Ennek a jelenségnek kissé regényes magyarázatát adja a következő *hóeke modell* (E.F. Moore 1961). Egy kör alakú pályán hóeke megy körbe.



A hó egyenletesen hull a pályára, a hóeke az előtte levő hómagassággal fordítottan arányos sebességgel halad. Megmutatható, hogy van a rendszernek stabil állapota, amihez a kezdő helyzetből konvergál. Ekkor minden pillanatban ugyanaz a P hómennyiség van a pályán, a hóeke sebessége állandó és a hó magassága lineárisan függ az ekétől való (pálya mentén mért) távolságtól. Ebből következik, hogy egy körben a hóeke $2P$ hómennyiséget takarít el.



A modellben a hó játssza a kulcsok szerepét. Az aktív kupac az eke jelenlegi helyzetétől az induló helyzetig menő ív, a maradék az alvó kupac. Így az egy körben eltakarított hó felel meg egy futamnak.

Minél előbbre tart a hóeke a pályán – vagyis minél nagyobb az aktív kupac minimális eleme –, annál nagyobb az esélye, hogy egy érkező hópehely a hóeke elé, és nem mögé esik. Másként mondva: a következő kulcs egyre növekvő eséllyel kerül az alvó kupacba.

Többfázisú (polifázisú) összefésülés

Vannak olyan k-rétű összefésülő sémák, melyek csak $k+1$ állományt használnak. A k input állományt egyetlen kimenő állományba fésüljük. Ha valamelyik input állomány üres lesz, akkor megállunk, és ez veszi át az output állomány szerepét. Itt csak a $k = 2$ esetre vetünk egy rövid pillantást. Ekkor 3 állományunk van, és minden fázisban kettő tartalmát fésüljük a harmadikra.

Nézzünk először egy példát. Legyen a három állomány f_1, f_2, f_3. Tegyük fel, hogy f_1 egy, f_2 pedig 5 ugyanolyan (mondjuk 1) hosszúságú futamból áll, f_3 üres. A következő táblázat szemlélteti a helyzetet. Az oszlopok adatai mutatják az összefésülő menetek utáni állapotot, az első oszlop a kezdő állapot. Egy bejegyzés első száma a futamok száma az állományban, a zárójelben levő érték pedig a futamhossz.

f_1	1(1)	$\emptyset$	1(3)	$\emptyset$	1(5)	$\emptyset$
f_2	5(1)	4(1)	3(1)	2(1)	1(1)	$\emptyset$
f_3	$\emptyset$	1(2)	$\emptyset$	1(4)	$\emptyset$	1(6)

Érezhetően túl sok mozgatást végzünk. Nem szerencsés a rekordok eloszlása a két állományban. Megmutatható, hogy ez a módszer akkor lesz a leggyorsabb, ha a futamok száma a két input állományban két szomszédos *Fibonacci-szám*. Először – emlékeztetőül – néhány hasznos tény a Fibonacci-számokról.

A Fibonacci-számokat a következő rekurzióval definiálhatjuk:

$$F_0 = 0, F_1 = 1, \quad F_{i+1} = F_i + F_{i-1}, \text{ ha } i \geq 1.$$

Hasznos az alábbi általános alak (indukcióval igazolható):

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right).$$

Innen látjuk, hogy F_n az $\frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n$ számhoz legközelebbi egész. Legyen $\phi = \frac{1+\sqrt{5}}{2}$ (az aranymetszés aránya).
A $\phi^2 = \phi + 1$ egyenlőségből adódik $n \geq 2$ esetén, hogy

$$\phi^{n-2} \leq F_n \leq \phi^{n-1}.$$

Ezután nézzük, miként alakul a futamok száma, ha a kezdő futamszámok F_{n-1} és F_n. A táblázatban most nem tüntetjük fel a futamok hosszát.

f_1	F_{n-1}	$\emptyset$	F_{n-2}	F_{n-4}	...
f_2	F_n	F_{n-2}	$\emptyset$	F_{n-3}	...
f_3	$\emptyset$	F_{n-1}	F_{n-3}	$\emptyset$	...

Kezdetben a futamok száma összesen $N = F_{n+1}$. Ebből $\phi^{n-1} \leq N$ és $n-1 \leq \log_\phi N = \log_2 N / \log_2 \phi \approx 1.44 \log_2 N$ adódik. A szükséges menetek száma ezért legfeljebb $1.44 \log_2 N + 1$. Ez valamivel rosszabb ugyan, mint a 4 állományt használó összefésülésé, de még mindig elég jó.

3.

Keresőfák

Vagy találunk utat, vagy építünk egyet.
HANNIBÁL

Az egyik legalapvetőbb adatkezelési igény, hogy adatok véges halmazait *hatékonyan* tárolni tudjuk. Hatékonyságon elsősorban az értendő, hogy a *keresés*, *beillesztés*, *törlés* és *módosítás* műveletei gyorsak legyenek. Ebben a fejezetben a problémának azzal a fontos speciális esetével foglalkozunk, amikor a tárolt S halmaz elemei egy $(U, <)$ rendezett típusból valók. Ebben az esetben további természetes igények is felmerülnek: érdekelhet bennünket az S legkisebb vagy éppen legnagyobb eleme, esetleg az S-nek az $[a, b]$ intervallumba eső elemei $(a, b \in U)$.

Az így körvonalazott feladatcsokornak megfelelő adatszerkezet a *keresőfa*, amihez egyetlen $(U, <)$ rendezett típus tartozik. A szerkezet célja az U egy véges S részhalmazát tárolni úgy, hogy BESZÚR, TÖRÖL, KERES, MIN, MAX, TÓLIG hatékonyak legyenek.

Ebben a megközelítésben jelentős egyszerűsítés van. A legtöbb esetben a tárolni kívánt adataink összetett szerkezetűek, több különböző típusú információelemből állnak. Ezek közül általában egy vagy csak néhány adja a keresés/tárolás alapját jelentő kulcsot. A terítékre kerülő algoritmusok szempontjából azonban eléggé közömbös, hogy a kulcs mellett vannak-e még további adatmezők is. A legtöbb esetben mit sem vesztünk azzal, ha feltesszük, hogy csupán U bizonyos elemeit kell tárolnunk. Ezzel az egyszerűsítéssel lehetővé válik, hogy csak az algoritmikus szempontból lényeges tényezőkre figyeljünk.

A műveletek pontos értelmezésében – elsősorban a kivételek kezelésénél – többféle változattal találkozhatunk. Ezeket nem célunk itt áttekinteni; csak a műveletekhez kapcsolódó jellegzetes teendőkről ejtünk szót.

A BESZÚR(x, S) eljárás(hívás) beilleszti az $x \in U$ elemet a tárolt S halmazba. TÖRÖL(x, S) hatására x kikerül az S halmazból. KERES(x, S) megadja

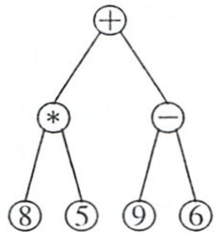
az x elem helyét az S-ben, feltéve, hogy $x \in S$. MIN(S) megadja az S-nek a $<$ rendezés szerinti legkisebb, MAX(S) pedig a legnagyobb elemét. Az intervallumkeresésnek az S halmaz mellett van még két paramétere. Ezek az U elemei; jelöljük őket a-val és b-vel. TÓLIG(a, b, S) eredménye azon $s \in S$ elemek összessége, melyekre teljesül, hogy $a \leq s \leq b$.

Az adatszerkezet nevében a *fa* szó arra hivatott utalni, hogy a legjobb ismert implementációk fákra épülnek, faszerű struktúrákat használnak. Mielőtt ezekből a megoldásokból ismertetnénk egy csokorra valót, egy kis kitérőt teszünk, rögzítve néhány bináris fákra vonatkozó tényt.

3.1. Bináris fák

Bináris fákkal már találkoztunk az előző fejezetben a kupac adatszerkezet kapcsán. Ott érintettük egy tárolási módjukat is, ami csak bizonyos speciális, terebélyes fákra működik. Itt egy általánosan használható megadási módot szeretnénk bemutatni. Ebben a fa egy csúcsa összetett szerkezetű; úgy gondolhatunk rá, mint egy rekord-típusú objektumra mondjuk a Pascal nyelvből. Az x csúcs fontosabb összetevői, „mezői" a következők: $elem(x)$ az x csúcsban tárolt érdemi információ; ez a későbbiekben legtöbbször egy U-beli kulcs lesz. A $bal(x)$ mező egy mutató az x csúcs bal fiára, $jobb(x)$ pedig a jobboldali fiára. A módszerek tárgyalásakor ezeket a mutatókat olykor azonosítani fogjuk a mutatott csúccsal. Nagyjából ennyi tartozik a bináris fák *alapvető reprezentáció*jához. Bizonyos esetekben további mezők is hasznosak lehetnek. Ilyen például az x csúcs apjára mutató $apa(x)$ mező vagy az x-ből és leszármazottaiból álló x-gyökerű részfa csúcsainak számát tartalmazó $s(x)$ mező.

Példa: A következő ábra egy bináris fát mutat; a tárolt elemeket - ezek számok és műveleti jelek - a csúcsokba írtuk. Jelölje x a fa gyökerét, y pedig a 9-et tartalmazó csúcsot. Érvényesek a következők: $bal(jobb(x)) = y$, $apa(apa(y)) = x$, $elem(bal(x)) = *$ és $s(x) = 7$.



A fejezetben fontos szerep jut majd a fák alakjának. Mint látni fogjuk, azok a fák lesznek hasznosak, amelyek eléggé telt alakúak. Ebből a szempontból azok

a fák ideálisak, melyeknek a szintjeik számához képest a lehető legtöbb csúcsuk van. Ha a szintek száma l, akkor ez $2^l - 1$ csúcsot jelent. Az ilyen fákat ebben a részben *teljes fáknak* fogjuk nevezni. Ez némiképp eltér a kupac adatszerkezetnél bevezetett használattól. A terminológiai kettősséget azért vállaljuk, mert eléggé általános a szakirodalomban.

A mostani meghatározás szerinti teljes fák úgy jellemezhetők, hogy minden levelük ugyanazon a szinten helyezkedik el, és minden belső csúcsuknak két fia van.

A bináris fák alkalmazásainál fontos szerepet játszanak az olyan módszerek – *bejárások* – amelyek valamilyen értelmes sorrendben végigmennek a fa csúcsain. Általában arról van szó, hogy amikor a bejárás során az y csúcs kerül sorra, akkor valami y-nal kapcsolatos teendőt végzünk el (pl. kinyomtatjuk a benne tárolt információt). A bejárásokat e teendőtől függetlenül érdemes tárgyalni, ezért egyszerűen csak azt mondjuk, hogy *meglátogatjuk* y-t. A legismertebb bejáró módszerek a *preorder*, *inorder* és *postorder* bejárások. Itt következik a rekurzív definíciójuk; x jelöli a fa gyökerét, az eljárások nevei pedig pre(), in() és post().

pre(x)	in(x)	post(x)
begin	**begin**	**begin**
látogat(x);	in(*bal*(x));	post(*bal*(x));
pre(*bal*(x));	*látogat*(x);	post(*jobb*(x));
pre(*jobb*(x))	in(*jobb*(x))	*látogat*(x)
end	**end**	**end**

A három eljárás csak a *látogat*(x); helyében különbözik egymástól. A preorder bejárás először meglátogatja a gyökeret, majd rekurzíve hívja önmagát először a bal, majd a jobb részfára. Az inorder bejárásnál a látogatás a két önhívás közé került, a postorder-módszernél pedig a hívások mögé.

Írjuk le az előző példa fájában levő elemeket abban a sorrendben, ahogy az eljárások meglátogatják a csúcsokat!

preorder sorrend: $+ * 85 - 96$;

inorder sorrend: $8 * 5 + 9 - 6$, megfelelően zárójelezve: $(8 * 5) + (9 - 6)$;

postorder sorrend: $85 * 96 - +$.

Az egyes sorrendek a fa által meghatározott aritmetikai kifejezés különböző írásmódjainak felelnek meg. A középső, az inorder sorrend adja a szokásos írásmódot.

Nézzük most a módszerek költségét. Elég a preorder bejárásra szorítkozni; a másik kettő ugyanúgy kezelhető. Tekintsük egységnyinek a pre(y)-hívás kezdésével és befejezésével járó költségeket; itt y a fa egy tetszőleges csúcsa. Ugyancsak legyen 1 a *bal*(y) és *jobb*(y) mutatók mentén való lépés, valamint *látogat*(y)

időigénye. Jelölje $T(n)$ a módszer maximális költségét legfeljebb n-pontú fákra. Feltehetjük, hogy $T(0) = 0$ és a rekurzió alapján

$$T(n) \leq c + \max_{0 \leq i \leq n-1} \{T(i) + T(n-1-i)\}.$$

A második tag a két rekurzív hívás költsége, az első pedig a hívásokon kívüli munka időigényét becslő n-től független c állandó. Egyszerű indukcióval adódik innen, hogy $T(n) \leq cn$. Ez ugyanis nyilván igaz, ha $n = 0$. Nagyobb n-ekre pedig az indukciós feltevés szerint

$$T(n) \leq c + \max_{0 \leq i \leq n-1} \{ci + c(n-1-i)\} = c + c(n-1) = cn.$$

A bejárások tehát $O(n)$, azaz lineáris időben megvalósíthatók.

Feladat: Tegyük fel, hogy adott az l szintből álló f bináris fa y csúcsa. Javasoljunk $O(l)$ költségű módszert a preorder (inorder, postorder) sorrendben az y után következő csúcs megtalálására.

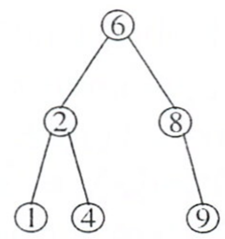
3.2. Bináris keresőfák, naiv algoritmusok

A keresőfa adatszerkezet megvalósításai közül első helyre kívánkoznak a *bináris keresőfák*. Ezek legfőbb közös vonása, hogy a tárolni kívánt $S \subseteq U$ halmaz elemei egy bináris fa csúcsaiban helyezkednek el, csúcsonként pontosan egy elem. Teljesül ezen felül a *keresőfa-tulajdonság*:

> *Tetszőleges x csúcsra és az x baloldali részfájában levő y csúcsra igaz, hogy $elem(y) \leq elem(x)$. Hasonlóan, ha z egy csúcs az x jobb részfájából, akkor $elem(x) \leq elem(z)$.*

Egy kényelmes megállapodás: a továbbiakban feltesszük, hogy nincsenek ismétlődő elemek a keresőfában. Ez a megállapodás jelentősen egyszerűsíti a módszerek leírását, a bennük levő gondolatok érzékeltetését. Az algoritmusok, amiket tárgyalni fogunk, kisebb-nagyobb módosításokkal működnek akkor is, ha megengedjük az ismétlődéseket. A hatékonyságuk viszont romlik, ha bizonyos elemekből túl sok példány van; gondoljunk arra, hogy milyen használhatatlan lenne a telefonkönyv, ha mindenkinek ugyanaz lenne a neve.

A megállapodást figyelembe véve a keresőfa-tulajdonság követelményeiben szigorú egyenlőtlenségeket érthetünk: az x bal részfájában levő elemek kisebbek x eleménél, a jobb részfában levők pedig nagyobbak annál. A következő példában az elemek természetes számok, a rendezés a szokásos. A fában tárolt S halmaz $\{1, 2, 4, 6, 8, 9\}$.



Feladat: Igazoljuk, hogy egy bináris keresőfa elemeit a fa inorder bejárása *növekvő sorrendben* látogatja meg.

Keresés bináris keresőfákban

Tegyük fel, hogy az S bináris keresőfának n csúcsa és l szintje van, és legyen $u \in U$. A KERES(s, S) első lépésében összehasonlítjuk S gyökerének s' elemét s-sel. Ha $s = s'$, akkor megtaláltuk a keresett csúcsot (elemet). Ha $s < s'$, akkor a keresőfa-tulajdonság miatt a bal, ha $s > s'$, akkor a jobb részfába megyünk tovább, ismételve a fenti lépést. Az előző példa fájánál a KERES$(4, S)$ a gyökérnél balfelé lép, hiszen $4 < 6$, utána jobbra, mert $4 > 2$; végül megtaláljuk a keresett elemet. Ugyanezt az utat járjuk be a KERES$(5, S)$ kapcsán. A négyest tartalmazó levélnél kiderül, hogy a kérdéses elem nincs a fában, hiszen eddig nem találtuk meg, és már nem tudunk továbblépni.

Egy összehasonlítás és egy mutató mentén megtett lépés árán az eredetinél egy szinttel alacsonyabb fában folytathatjuk a keresést. Ebből következik, hogy a módszer költsége arányos a szintek számával: $O(l)$.

A MIN és MAX műveletek is eléggé egyszerűek. Az előbbinél addig megyünk a gyökértől balra a fában, amíg lehetséges, az utóbbinál pedig a jobboldali irányt tüntetjük ki figyelmünkkel. E két eljárás költsége is $O(l)$.

A TÓLIG(a, b, S) hívás az S halmaz a és b közé eső kulcsait hivatott összegyűjteni. Evégből KERES(a, S) segítségével megkeressük a szóban forgó intervallum elejét, majd inorder sorrendben ellépdelünk az utolsó olyan kulcsig, ami még nem nagyobb, mint b. Az inorder lépkedéshez hasznos a csúcsokban egy mutató – úgynevezett inorder fonal – az inorder bejárás szerinti következő csúcsra. Ha vannak ilyen mutatóink, akkor TÓLIG(a, b, S) költsége $O(l + k)$, ahol k az S a és b közé eső elemeinek a száma. Ha nincsenek inorder fonalaink, akkor inorder bejárással $O(n)$ időben kaphatjuk meg a kívánt eredményt.

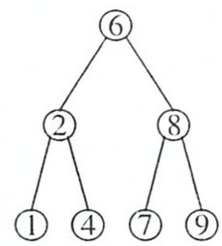
Naiv módszerek beszúrásra és törlésre

A különféle bináris keresőfa-típusok között a döntő különbség a beszúrás és a törlés algoritmusaiban van. Itt a legegyszerűbb, az úgynevezett naiv módszereket mutatjuk be. A BESZÚR(s, S) és TÖRÖL(s, S) végrehajtása is egy KERES(s, S)

hívással kezdődik. Beszúráskor a beillesztendő s elem helyét kell megkeresni S-ben, a TÖRÖL(s, S) esetén pedig a törlendő s elemet.

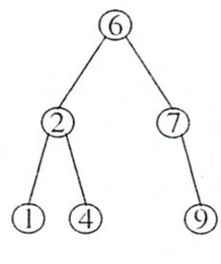
BESZÚR(s, S) végrehajtásakor a keresés után nincs teendőnk, ha már van olyan x csúcsa a fának, melyre $elem(x) = s$. Ellenkező esetben a keresés úgy ér véget, hogy nem tudunk továbblépni a csúcsból, ahol vagyunk. A megfelelő (jobb, vagy bal) fiú nincs a fában. A beszúrást úgy végezzük el ezután, hogy ezt a hiányzó fiút mint új levelet a fához adjuk. Az új csúcs eleme s lesz. Könnyű meggondolni, hogy ezután is érvényben marad a keresőfa-tulajdonság. Az eljárás költsége $O(l)$.

Példa: Az előző S fába illesszük be új elemként a hetest, vagyis hajtsuk végre a BESZÚR$(7, S)$ utasítást. A kereső fázisban arra jutunk, hogy a hetesnek a nyolcast tartalmazó csúcs bal fiában volna a helye. Ilyen fiú nincs a fában, ezzel bővíteni kell a szerkezetet. Az eredmény:



Ami a törlést illeti, nincs érdemi teendőnk, ha a szóban forgó s elem nincs a fában. Ellenkező esetben jelölje x az S-nek azt a csúcsát, amiben s ül. Ezek után TÖRÖL(s, S) könnyű, ha a szóban forgó x csúcsnak legfeljebb egy fia van. Ekkor elegendő, hogy x-et a fiával helyettesítjük: $x \leftarrow$ fiú(x). Ha viszont x-nek két fia van, akkor ez az út nem járható; csak egyik fiút tehetnénk x helyére, a másik árván maradna. A megoldás az lesz, hogy a feladatot visszavezetjük egy könnyű törlésre. Jelölje y azt a csúcsot, amiben az x bal részfájának a maximális eleme van. Az x ismeretében y-t MAX$(bal(x))$ $O(l)$ költséggel megadja. A recept ezután egyszerű. Az y-ban levő s' elemet tesszük s helyére x-be, majd töröljük az y csúcsot. Utóbbi egy könnyű törlést jelent, mert y-nak nem lehet jobboldali fia. Azt is könnyű meggondolni, hogy az így módosult fára érvényben marad a keresőfa-tulajdonság. Az s' definíciója miatt kisebb az x jobb részfájának minden eleménél, és nagyobb a bal részfa megmaradó elemeinél. Az összköltség ekkor is $O(l)$.

Példa: Az előző rajzon látható S fából töröljük a nyolcast. Ez nehéz törlésnek minősül, hiszen a szóban forgó x csúcsnak két fia van. Az x bal részfájának maximális (és egyetlen) eleme 7. Ez kerül x-be; a hetest tartalmazó csúcsot pedig kitöröljük:



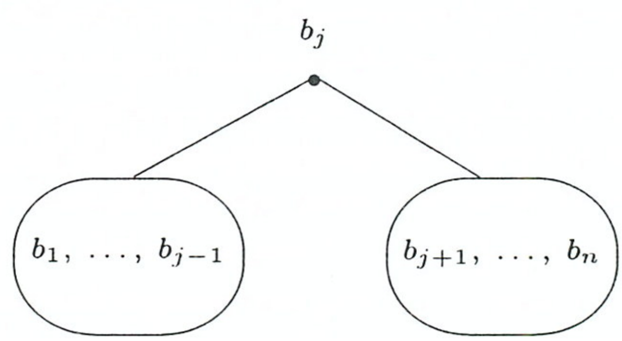
Naiv beszúrásokkal épített bináris fák átlagos költsége

Ha egy bináris fának l szintje van, akkor a csúcsok számára $n \leq 1+2+2^2+\cdots+2^{l-1} = 2^l - 1$ adódik, amiből $l \geq \log_2(n+1)$. A keresés szempontjából tehát a legjobb – azaz legkisebb – szintszám, amit n-pontú fánál elérhetünk, körülbelül $\log_2 n$.

A következőkben érvet mutatunk amellett, hogy a naiv beszúrásokkal épített fák átlagos értelemben nem rosszak; egy beillesztés átlagosan $O(\log_2 n)$ összehasonlításba kerül. A pontos modell a következő: tegyük fel, hogy az U univerzum $b_1, < b_2 < \ldots < b_n$ elemeiből bináris keresőfát építünk. A b_i elemeket egy véletlen $a_1, \ldots, a_n$ sorrendben szúrjuk be a naiv módszerrel a kezdetben üres fába. Legyen $T(n)$ a beszúrások során fellépő összehasonlítások átlagos száma. A $T(n)$ mennyiséggel szeretnénk méri a fa építésének átlagos költségét. Az átlagot az elemek $n!$ lehetséges érkezési sorrendjére vesszük. Valószínűségi terminológiát használva úgy is fogalmazhatunk, hogy a $b_1, b_2, \ldots, b_n$ elemek minden érkezési sorrendje egyenlően valószínű. Ekkor tetszőleges j-re $\frac{1}{n}$ az esélye, hogy $a_1 = b_j$, hiszen mindegyik elem éppen ugyanannyi (nevezetesen $(n-1)!$) érkezési sorrendnél lesz első. Mindez azt jelenti, hogy

$$(*) \qquad T(n) = \frac{1}{n}\sum_{j=1}^{n} (\text{az olyan fa átlagos költsége, melyre } a_1 = b_j).$$

Egy fa költségén a felépítéséhez használt összehasonlítások számát értjük. Jelölje $F(j)$ a zárójelben levő tagot. Ha $a_1 = b_j$, akkor a keresőfa-tulajdonság miatt végül az a_1 bal részfájában $j-1$, a jobb részfában pedig $n-j$ elem lesz.



Itt a baloldali részfán belüli összehasonlítások száma $T(j-1)$ lesz, amihez hozzá kell adnunk a b_j gyökérrel való összehasonlításokat. A bal részfa átlagos költsége tehát $j-1+T(j-1)$. Hasonló meggondolással a jobb részfa költsége $n-j+T(n-j)$. A kettőt összegezve kapjuk, hogy $F(j)=j-1+T(j-1)+n-j+T(n-j)=n-1+T(j-1)+T(n-j)$. Mindezeket visszaírhatjuk a $(*)$ formulába. Használva azt is, hogy az üres fa ingyen van, azaz $T(0)=0$, $n>0$-ra a következő rekurziót nyerjük:

$$T(n)=n-1+\frac{2}{n}\sum_{j=0}^{n-1}T(j).$$

Vegyük észre, hogy ez megegyezik a gyorsrendezés elemzésekor kapott $(+)$ rekurzióval. Az ott taglalt módon adódik tehát, hogy

$$T(n)<2n\log n+O(n)\approx 1,39\log_2 n+O(n).$$

Egy elem beszúrásához átlagosan tehát legfeljebb $1,39\log_2 n$ összehasonlítás elegendő. A becslés érvényes a keresés átlagos költségére is. Hasonló módszerekkel megmutatható, hogy az átlagos szintszám (azaz l várható értéke) is $O(\log n)$. Ez nem rossz, ha figyelembe vesszük, hogy adott n mellett a legjobb, amit elérhetünk $l=\log_2(n+1)$. Ez akkor teljesül, ha a fa teljes (l szintje és összesen 2^l-1 csúcsa van).

3.3. 2-3-fák

Ebben a szakaszban egy igen hatékony keresőfa-konstrukciót mutatunk be. Ennek is fa a tárolási szerkezete, de a binárisnál valamivel bonyolultabb; egy nem-levél csúcsnak 2 vagy 3 fia lehet. Innen származik a szerkezet elnevezése. Egy kulcs (U eleme) a fa több csúcsában is előfordulhat, és egy csúcsban egy vagy két kulcs is lehet. Úgy érdemes elképzelni a helyzetet, hogy U elemei egy rekordtípus kulcsai (pl. személyi szám egy személyi adatokat tartalmazó rekordban). A cél a rekordok hatékony elérése. Maguk a rekordok a fa legalsó szintjén helyezkednek el, a fa belső csúcsai csak kulcsokat és mutatókat tartalmaznak (kivéve, ha csak egy rekordunk van). A belső csúcsokban levő kulcsok a levelekben tárolt rekordok kulcsai közül kerülnek ki. Szerepük az, hogy jelzőtáblák módjára segítsék a fában való közlekedést.

A *2-3-fa* egy (lefelé) irányított gyökeres fa, melyre igazak az alábbiak:
1. A tárolni kívánt rekordok a fa leveleiben helyezkednek el, a kulcs értéke szerint balról jobbra növekvő sorrendben. Egy levél egy rekordot tartalmaz.

2. Minden belső (azaz nem levél) csúcsból 2 vagy 3 él megy lefelé; ennek megfelelően a belső csúcsok egy illetve két $s \in U$ kulcsot tartalmaznak. A belső csúcsok szerkezete tehát kétféle lehet. Az egyik típus így ábrázolható:

m_1	s_1	m_2	s_2	m_3

Itt m_1, m_2, m_3 mutatók a csúcs részfáira, s_1, s_2 pedig U-beli kulcsok, melyekre teljesül, hogy $s_1 < s_2$. Az m_1 által mutatott részfa minden kulcsa kisebb, mint s_1, az m_2 részfájában s_1 a legkisebb kulcs, és minden kulcs kisebb, mint s_2. Végül m_3 részfájában s_2 a legkisebb kulcs. Előfordulhat, hogy egy csúcsból az utolsó két mező hiányzik. A belső csúcsoknak ez a típusa valamivel egyszerűbb alakú:

m_1	s_1	m_2

Ez esetben a csúcsnak csak két részfája van; a baloldaliban vannak az s_1-nél kisebb kulcsú rekordok, a jobboldaliban pedig s_1 a legkisebb kulcs.
3. A fa levelei a gyökértől egyforma távolságra vannak.

A következő állítás a műveletek költségének becslésekor lesz hasznos. Lényegében azt mondja, hogy a fa szintjeinek száma barátságosan alacsony a fában levő rekordok számához mérten.

Állítás: *Ha a fának m szintje van, akkor a levelek száma legalább 2^{m-1}. Megfordítva, ha $|S| = n$ (itt $S \subseteq U$ a fában tárolt kulcsok halmaza; $|S|$ megegyezik a tárolt rekordok számával), akkor $m \leq \log_2 n + 1$.*

Bizonyítás: A 3. tulajdonság szerint a fa i-edik szintjén levő csúcsok mind belső csúcsok, ha $1 \leq i \leq m - 1$. A 2. tulajdonság szerint minden ilyen csúcsnak legalább két fia van. Ezekből könnyű indukcióval adódik, hogy az i-edik szinten legalább 2^{i-1} csúcs van ($1 \leq i \leq m$). Ezt $i = m$-re alkalmazva $2^{m-1} \leq n$, amiből logaritmusokat véve $m - 1 \leq \log_2 n$. □

Keresés 2-3-fákban

A kulcsoknak a részfákban való ilyen elosztása lehetővé teszi, hogy a bináris fáknál megismert módszerhez hasonlóan végezzük a keresést. Legyen S egy 2-3-fa, és $s \in U$. A KERES(s, S) első lépéseként az s kulcsot összehasonlítjuk az S gyökerében levő s_1 és esetleg s_2 kulcsokkal. Ennek eredményeképpen megtudjuk, hogy a három (vagy két) részfa közül melyikben kell folytatni a keresést. Ha $s < s_1$, akkor az m_1 mutató mentén találjuk meg az érdekes részfa gyökerét. Ha $s_1 \leq s < s_2$, akkor a második mutatót követjük. Ha pedig $s_2 \leq s$, akkor m_3

mutatja a helyes irányt. Természetesen, ha s_2 és m_3 hiányzik, akkor csak két eset van: $s < s_1$, illetve $s_1 \leq s$.

A keresést aztán ugyanígy folytatjuk egy szinttel lejjebb. Legyen n a fában tárolt rekordok száma, m pedig a fa szintjeinek a száma. Az elmondottak szerint szintenként nem több, mint két kulcs-összehasonlítással, összesen tehát legfeljebb $2m$-mel kideríthetjük, hogy a keresett rekord benne van-e a fában. Igenlő válasz esetén meg is találjuk az s kulcsú rekordot. Az állítást figyelembe véve látjuk, hogy az összehasonlítások száma legfeljebb $2(\log_2 n + 1)$; a többi költség arányos ezzel. Így a keresés összköltségére igen kedvező korlátot kapunk: $O(\log n)$. A korlát *minden esetben* (tehát nem csak átlagos értelemben) érvényes.

Beszúrás és törlés

A 2. és 3. követelmények igen szigorúan szabályozzák a 2-3-fák alakját. Ennek a két tulajdonságnak köszönhető a versenyképes keresési idő. A beszúrás és törlés kapcsán a fő gondot az jelenti, hogy megőrizzük érvényességüket. A módszerek lényeges ötleteit példákon keresztül mutatjuk be. Ennek során feltesszük, hogy U elemei nagybetűk: $A, B, C \ldots$, a szokásos rendezéssel.

A BESZÚR(s, S) végrehajtása kereséssel kezdődik, és csak akkor van érdemi munka, ha nincs a fában s kulcsú levél. Tegyük fel, hogy ez a helyzet, és jelölje x a legalsó belső csúcsot a kereső út mentén. Tegyük fel, hogy x így néz ki:

m_1	J	m_2

Az m_1 által meghatározott levél kulcsa legyen C (a másik levélé szükségképpen J). Ha például $s = K$, akkor x így módosul:

m_1	J	m_2	K	m_3

Az m_3 mutató fog az új levélre mutatni. Ha viszont $s = I$, akkor x tartalma a következő lesz:

m_1	I	m_3	J	m_2

Ha s a legkisebb kulcsot is megelőzi, mondjuk $s = A$, akkor x így alakul:

m_3	C	m_1	J	m_2

Az új rekord beillesztése tehát elég egyszerű, ha x-nek csak két fia van.

Igazán érdekessé akkor válik a helyzet, ha a beszúrás előtt x már két kulcsot tartalmaz. Tegyük fel, hogy x a legutóbbi ábra szerinti állapotban van, és nézzük,

mi a teendő, ha most az $s = H$ kulcsú rekordot kell beilleszteni. Ekkor alkalmazzuk a *csúcsvágás* elnevezésű elegáns ötletet: az x mellé egy további y csúcsot veszünk. A két csúcs tartalma

$$\begin{array}{|c|c|c|}\hline m_3 & C & m_1 \\ \hline \end{array} \quad \text{illetve} \quad \begin{array}{|c|c|c|}\hline m_4 & J & m_2 \\ \hline \end{array}$$

lesz. Az m_4 fog a H kulcsú rekordra mutatni. Ezután az új y csúcsot is a fába kell illeszteni. Ez azt jelenti, hogy az x apjába kell egy új kulcs–mutató párt tennünk. A beillesztendő kulcs a három érintett (a két régi és az új) közül mindig a nagyság szerint középső. Ez a kulcs a jelen esetben H. Az egy szinttel feljebb történő beszúrást *ugyanazzal* a módszerrel végezzük, ahogy x-nél eljártunk. Tehát ha x apjának csak két gyermeke volt, akkor az egyszerű ágon fejezzük be a munkát. Ha nem, akkor itt is csúcsvágást alkalmazunk. A csúcsvágások sora elgyűrűzhet esetleg a fa gyökeréig. Mi ilyenkor a teendő? Tegyük fel, hogy az utolsó példában x a fa gyökere volt. (Ekkor az m_i mutatók esetleg nagyobb részfákra mutatnak.) A csúcsvágás után keletkező x és y rekordok fölé egy új gyökeret teszünk:

$$\begin{array}{|c|c|c|}\hline m & H & m' \\ \hline \end{array}$$

Itt m az x, m' pedig az y csúcsra mutat. A fa szintjeinek száma eggyel nőtt. A lényeges mozzanat ebben, hogy a növekedés a fa tetején történt; ezáltal megtartottuk a 2-3-fákkal kapcsolatos 3. követelményt. A gyökértől levélig menő utak mindegyike hosszabb lett. A második követelményről is gondoskodtunk: az eseteket végignézve az olvasó meggyőződhet arról, hogy sehol sem hagytunk belső csúcsot kettőnél kevesebb gyermekkel.

Nézzük ármost a TÖRÖL(s, S) végrehajtását. Legyen ismét x a legalsó belső csúcs a kereső út mentén. Ha x-nek három fia van, akkor az s kulcsú levél törlése után x-ben az értelemszerű változtatásokat elvégezve készen vagyunk. Gondot jelent viszont, ha x-nek csak két fia van. Ekkor a törlés után megsérülne a 2-3-fákkal szembeni 2. követelmény, hiszen x-nek csak egy fia maradna. Még mindig egyszerű a dolgunk, ha x (valamelyik) szomszédos testvérének 3 fia van; ekkor ugyanis ezek közül egyet áttehetünk x-be. Csak x-et, adakozó kedvű testvérét és az apjukat kell módosítani. Ezeket a módosításokat elvégezve ismét készen vagyunk.

Mindezek nem lehetségesek, ha x egyik szomszédos testvérének sincs három részfája. Ekkor használható a csúcsvágás fordítottját jelentő gondolat, a *csúcsösszevonás*. A csúcsösszevonás lényege, hogy x-et és az egyik testvérét egyetlen csúccsal helyettesítjük. Ennek a csúcsnak három fia lesz. A csúcsösszevonás kivitelezését is egy példán keresztül mutatjuk be. Legyenek x és a fában szomszédos testvére, y az alábbiak:

m_1	J	m_2		m_3	M	m_4

Legyenek az m_1, illetve m_3 által mutatott levelek kulcsai A és K. (Az m_2-höz és m_4-hez tartozó kulcsok szükségképpen J, illetve M.) Ha $s = K$ a törlendő kulcs, akkor az összevonás után x így néz ki.

m_1	J	m_2	M	m_4

Az y csúcsot töröljük. A K kulcsértéket és a hozzá tartozó mutatót törölni kell x apjából (törlés egy szinttel feljebb). A többi három eset (azaz A vagy J vagy M törlése) hasonlóan végezhető. Csúcsösszevonáshoz vezető törlés után egy szinttel magasabban is felmerül egy törlési igény. Ezt az itt ismertetett módon kezeljük. Ennek során is szükség lehet csúcsösszevonásra, ami törlést igényel a nagyapák szintjén; és így tovább. A beszúrásnál tapasztalt jelenséghez hasonlóan a törlések is elgyűrűzhetnek egészen a gyökérig. Külön figyelemre akkor van szükség, ha a törlés után a gyökérnek csak egyetlen fia maradna. Például tegyük fel, hogy a gyökér

m_1	L	m_2

alakú, és törölnünk kell az L kulcsot és a hozzá tartozó m_2 mutatót. Ekkor az m_1 által mutatott részfa gyökere lesz az új gyökér. A fa szintjeinek száma eggyel kevesebb lesz. A változás ismét a fa tetején történt; ebből könnyen következik, hogy a törlés algoritmusa is megtartja a 2-3-fákkal kapcsolatos követelményeket.

Mind a BESZÚR, mind a TÖRÖL költsége $O(\log n)$, hiszen munkát csak a kereső út csúcsain, illetve azok közvetlen szomszédain kell végezni; továbbá egy csúcsra konstans mennyiségű összehasonlítás és mezőmódosítás jut.

TÓLIG(a, b, S) megvalósítása hasonló a bináris fáknál tárgyalt megoldáshoz. Először megkeressük S-ben a keresési intervallumba eső legkisebb elemet; ennek költsége az előbbiek szerint $O(\log n)$, majd végiglépkedünk a felső korlátig. Ez utóbbi fázis a fa leveleinek növő sorrendű láncolásával segíthető. Az összköltség $O(\log n + d)$, ahol d az eredményül kapott rekordok száma.

Összegezésül elmondhatjuk, hogy a 2-3-fák a keresőfa adatszerkezet igen jó megvalósítását adják. A három fő művelet – a keresés, a beszúrás és a törlés – elvégezhető $O(\log n)$ költséggel, ahol n a tárolt rekordok száma. A 2-3-fák algoritmusai viszonylag egyszerűen programozhatók, és a gyakorlatban is kedvező viselkedést mutatnak.

3.4. B-fák

A B-fák és különböző változataik (B^*-fák, stb.) adják a keresőfa adatszerkezet ma ismert legjobb külső táras megvalósítását. A módszerek olyannyira fontosak és elfogadottak, hogy szabványok részeivé váltak. Ilyen megoldást ír elő egyebek között az IBM VSAM szabványa indexelt szekvenciális állományok létrehozására, kezelésére. A B-fákat és algoritmusaikat a feladat jelentőségéhez és a megoldások egyszerűségéhez képest meglehetősen későn fedezték fel (R. Bayer, E. McCreight, 1972).

A cél tehát egy rekordokból álló állomány tárolása külső tárban úgy, hogy az elérés alapjául egy $(U, <)$ rendezett halmazból való kulcsok szolgálnak. A kitüntetett műveletek itt is a következők: BESZÚR, TÖRÖL, KERES, MIN, MAX, TÓLIG. Ezeket szeretnénk hatékonyan megvalósítani.

A külső tárak alapvető sajátosságairól már szóltunk a rendezés kapcsán. A tárat lapokba szervezettnek fogjuk elképzelni. Ennek folyománya például, hogy a fák csúcsai is lapok lesznek. A költségtényező pedig, amivel a hatékonyságot mérjük, a lapelérések száma.

A B-fák a 2-3-fák természetes általánosításai. A lényegi különbség abban van, hogy a B-fák belső csúcsainak háromnál több fia is lehet. Kevésbé fontos különbség, hogy a fa egy levelében (amit egy lapnak képzelhetünk el) egynél több rekord is helyet kaphat. Legyen $m \geq 3$ egy egész.

Egy *m-edrendű B-fa*, röviden B_m-fa egy gyökeres, (lefelé) irányított fa, melyre érvényesek az alábbiaknak:

(a) A gyökér foka legalább 2, kivéve esetleg, ha a fa legfeljebb kétszintes.

(b) Minden más belső csúcsnak legalább $\lceil \frac{m}{2} \rceil$ fia van.

(c) A levelek a gyökértől egyforma messze vannak.

(d) Egy csúcsnak legfeljebb m fia lehet.

A 2-3-fák esetéhez hasonlóan a tárolni kívánt rekordok itt is a fa leveleiben helyezkednek el; egy levélben a lapmérettől és a rekordhossztól függően több rekord is lehet. A leveleket jelentő lapok balról jobbra, kulcsérték szerint növekvő sorrendben láncolva helyezkednek el.

A B-fák belső csúcsai is hasonlítanak a 2-3-fák belső csúcsaira. Az eltérés annyi, hogy itt több bejegyzés lehetséges. Egy belső csúcs így néz ki:

m_0	s_1	m_1	s_2	m_2	$\ldots$	s_i	m_i

ahol a (b) és (d) követelményeknek megfelelően $\lceil \frac{m}{2} \rceil - 1 \leq i \leq m - 1$. Korábbi jelöléseinkkel összhangban az $s_i \in U$ egy kulcs, m_j pedig mutató egy részfa gyökerére. Az m_j mutató által meghatározott részfa minden s kulcsára igaz, hogy

$s_j \leq s$ és $s < s_{j+1}$. Az m_0, illetve m_i részfák kulcsaira csak egy feltétel van: az elsőben a kulcsoknak kisebbeknek kell lenniük, mint s_1, az utolsóban pedig legalább akkoráknak, mint s_i. Előírható még az is (bár az algoritmusok működéséhez nem feltétlenül szükséges), hogy $j > 0$ esetén m_j részfájában a minimális kulcs s_j legyen. A fának azon szintjeit, ahol a belső csúcsok helyezkednek el, *indexszinteknek* szokás nevezni.

A meghatározó tulajdonságokból közvetlenül látszik, hogy a B_3-fák lényegében a 2-3 fa. A B-fák a 2-3-fák természetes általánosításainak tekinthetők. Ahogy már utaltunk rá, a B-fákat elsősorban külső táras adatszerkezetként alkalmazzák, és ebből adódóan egy csúcsnak általában egy lapot foglalnak le. Így m aktuális értéke a lapméret, a kulcshossz és a mutatók hosszának függvénye. Ugyanakkor az is gyakran előfordul, hogy egy levélben több, mint egy rekord helyezkedik el.

Tegyük fel, hogy egy B-fának n levele és k szintje van, és keressünk összefüggést e két paraméter között. Az érdektelen kicsi fáktól eltekintve a gyökérnek legalább két fia van, a többi belső csúcsnak pedig legalább $\lceil \frac{m}{2} \rceil$. A 2-3-fákra vonatkozó állításhoz hasonlóan kapjuk ezekből, hogy $n \geq 2\lceil \frac{m}{2} \rceil^{k-2}$, amiből $\log_{\lceil \frac{m}{2} \rceil} \frac{n}{2} + 2 \geq k$. Innen kettes alapú logaritmusra térve

$$k \leq \frac{\log_2 n - 1}{\log_2 \lceil \frac{m}{2} \rceil} + 2.$$

Keresés során itt is a gyökértől haladunk a levelek felé; az aktuális csúcsban levő kulcsokkal való összehasonlítások mondják meg, hogy melyik részfába kell továbblépni. Vegyük észre, hogy k éppen a kereséshez szükséges lapelérések száma, feltéve, hogy minden lap külső tárban van. (Nem szokatlan gyorsítási ötlet a fa felső néhány szintjének a belső memóriában való tárolása.) A kapott korlát mutatja, hogy nagy m érték (nagy elágazási tényező) az előnyös.

Például, ha $m = 32$, $n = 2^{20}$ (itt n az alsó szint *lapjainak* száma), akkor $k \leq \frac{19}{4} + 2 < 7$. Egy rekord keresése tehát legfeljebb 6 lap elérését igényli. Megjegyezzük még itt, hogy a becslésben úgy vettük, hogy a lapok éppen a (b) feltételbeli minimális mértékben vannak kitöltve. A gyakorlatban ez a szélsőséges helyzet meglehetősen ritkán fordul elő; a tényleges keresési idő valamivel kedvezőbb a becsültnél.

A további keresőfa-műveletek (MIN, MAX, TÓLIG, BESZÚR , TÖRÖL) is a 2-3-fák műveleteinek kézenfekvő általánosításai. Így például a beszúrásnál csúcsvágást, a törlésnél csúcsösszevonást alkalmaznak. Ezeket a technikákat ebben az összefüggésben *lapvágásnak*, illetve *lapösszevonásnak* nevezzük. Az algoritmusok részleteit mellőzzük; csak annyit jegyzünk itt meg, hogy a költségük – a TÓLIG kivételével – ez esetben is arányos a szintek számával.

Példa: A B_{11}-fák esetén keresztül szeretnénk érzékeltetni, hogy a (b) és (d) feltételek tényleg megtarthatók lapvágás/összevonás során. Ekkor ugyanis $m = 11$ és $\lceil \frac{m}{2} \rceil = 6$. Lapvágásra akkor kerül sor, ha egy olyan csúcsba kell új gyereket illeszteni, amelyben már 11 mutató van. A vágás után az összesen 12 gyerek 2 hatgyermekes csúcsot fog alkotni, tehát (b) éppen teljesül.

Lapösszevonásra akkor kényszerülünk, ha egy olyan hatgyermekes csúcsból kell törölni, amelynek a szomszédos testvéreiben is csak hat mutató van. A törlés és összevonás után két csúcs helyett egyet kapunk; ebben 11 mutató lesz.

3.5. *AVL*-fák

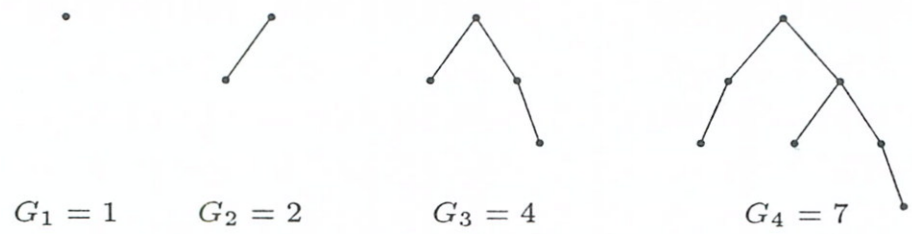
Egy időre visszatérünk a bináris keresőfákhoz. Korábban láttuk, hogy a keresés és a többi művelet sebességének szemszögéből a lényeges jellemző a fa *szintjeinek száma*, más szóval *magassága*. Minél kisebb a magasság, annál jobb becslés adható a keresés idejére. Mi ezt a legkedvezőtlenebb esetekre állapítottuk meg, de hasonló a helyzet az átlagos viselkedéssel is. Egy fa magassága akkor mondható jónak, ha legfeljebb $c \log_2 n$, ahol n a csúcsok száma és c egy kis pozitív állandó. A legterebélyesebb fáknál ez a c érték 1 körül van: gondoljunk a teljes fákra, amelyek magassága $\log_2(n+1)$. Az olyan fa-konstrukciókat, amelyeknél c 1-nél nem sokkal nagyobb, *kiegyensúlyozott* fáknak nevezzük. A c értéket illetően két ellentétes irányú tényezőt kell figyelembe vennünk. A keresés szempontjából a kisebb c a jobb; ugyanakkor nagyobb c-vel a feltétel könnyebben teljesíthető, algoritmikusan kényelmesebben elérhető.

Az első kiegyensúlyozott keresőfa-konstrukciót, amit szemügyre veszünk, *AVL*-fának nevezik. Az elnevezés a szerkezet szerzőinek névbetűiből alkotott mozaikszó (G. M. Adelszon-Velszkij, E. M. Landisz, 1962). Hasznos lesz a következő: jelölje $m(f)$ az f bináris fa magasságát (szintjeinek számát). Legyen x az f fa egy csúcsa; ekkor $m(x)$ jelöli az x-gyökerű részfa magasságát.

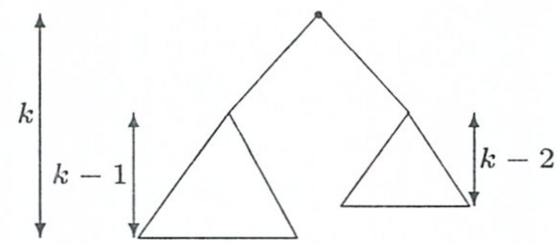
Definíció (*AVL*-tulajdonság): *Egy bináris keresőfa AVL-fa, ha minden x csúcsára teljesül, hogy*

$$|m(bal(x)) - m(jobb(x))| \leq 1.$$

A feltétel szerint egy csúcs jobb és bal részfájának a magassága közel egyenlő. Most megmutatjuk, hogy az előírás tényleg egy kiegyensúlyozottsági feltétel. Látszólag messziről kezdjük: legyen G_k a k magasságú (szintszámú) *AVL*-fák minimális csúcsszáma. Próbáljuk meghatározni G_k nagyságrendjét! Az első néhány k-ra könnyen kapjuk a pontos értékeket: $G_1 = 1$, $G_2 = 2$, $G_3 = 4$ és $G_4 = 7$.



A rajzon látható szabályszerűség általában is érvényes. A k szintszámú minimális csúcsszámú *AVL*-fa gyökerének egyik részfája $k - 1$, a másik $k - 2$ szintű; az eredeti fa minimalitása miatt pedig mindkét részfa minimális csúcsszámú.



Innen $k \geq 3$ esetén az alábbi rekurziót nyerjük:

$$G_k = 1 + G_{k-1} + G_{k-2}.$$

Emlékeztetjük az olvasót, hogy F_i jelöli az i-edik Fibonacci-számot. Érvényes a következő:

Tétel: $G_k = F_{k+2} - 1$ *ha* $k \geq 1$.

Bizonyítás: $k = 1$, 2 esetén az állítás nyilvánvaló. Ha pedig $k > 2$, akkor a rekurzió alapján indukciót használhatunk:

$$G_k = 1 + G_{k-1} + G_{k-2} = 1 + F_{k+1} - 1 + F_k - 1 = F_{k+2} - 1.$$

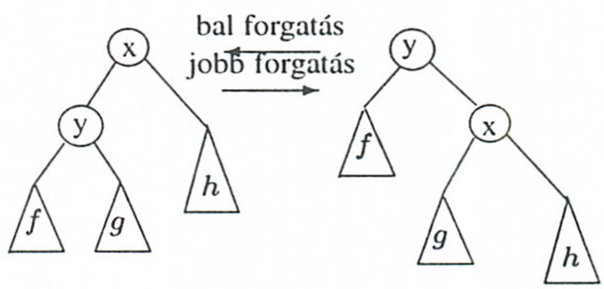
Az utolsó lépésben alkalmaztuk a Fibonacci-számokra teljesülő $F_{k+1} + F_k = F_{k+2}$ összefüggést. □

Következmény: *Egy n-pontú AVL-fa szintjeinek k száma nem több mint $O(\log n)$, pontosabban $k \leq 1.44 \log_2(n + 1)$.*

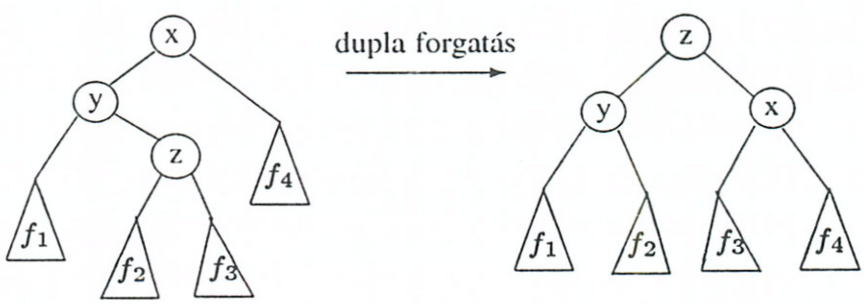
Bizonyítás: A tétel szerint $n \geq F_{k+2} - 1$. Innen a Fibonacci-számokra vonatkozó alsó becslésből $n + 1 \geq \phi^k$ és ezért $\log_\phi(n + 1) \geq k$, amiből $k \leq 1.44 \log_2(n + 1)$. □

Az *AVL*-fák tényleg tekinthetők kiegyensúlyozott fáknak. A kérdéses c érték 1.44 körül van. Úgy is fogalmazhatunk, hogy az *AVL*-fákban való keresés hatékony, mert a magasságuk legfeljebb 1.44-szer nagyobb az ugyanannyi csúcsból álló tökéletes alakú bináris fáénál.

A kérdés ezek után, hogy miként lehet az *AVL*-tulajdonságot megőrizni, karbantartani? Hogyan lehet a BESZÚR és TÖRÖL eljárásokat úgy megvalósítani, hogy megtartsák az *AVL*-tulajdonságot? Az alapvető eszköz a *forgatás*. Legyen S egy bináris keresőfa, melynek gyökere az x csúcs, ennek bal fia y. Jelölje f az y bal, g pedig a jobb részfáját. Legyen h az x jobb részfája. A bináris keresőfa-tulajdonság megmarad, ha az S fát átalakítjuk úgy, hogy y lesz a gyökere, ennek bal részfája marad f, az y jobb fia x, ennek részfái g (bal) és h (jobb). Ezt az átalakítást jobb forgatásnak nevezzük, az inverzét pedig bal forgatásnak.



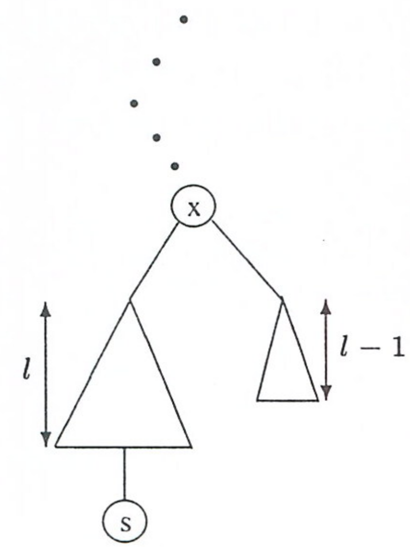
Legyen az S bináris keresőfa, gyökere az x csúcs, ennek bal fia y, y jobb fia z. Végezzünk el egy bal forgatást az y gyökerű részfára, majd egy jobb forgatást az így kapott S-re. Ennek az operációnak a neve dupla forgatás.



Dupla forgatásnak nevezzük ennek a lépéspárnak a tükörképét is, amikor y az x jobb fia, és z az y bal fia. A beszúrás és törlés megvalósítására a naiv algoritmusokat használjuk, kiegészítve azzal, hogy a művelet elvégzése után forgatásokkal visszaállítjuk (ha szükséges) az *AVL*-tulajdonságot.

Tétel: *Legyen S egy n csúcsból álló AVL-fa.* BESZÚR(s, S) *után legfeljebb egy (esetleg dupla) forgatással helyreállítható az AVL-tulajdonság. A beszúrás költsége ezzel együtt is* $O(\log n)$.

Bizonyítás: A korábbiakkal összhangban egy f fa (z csúcs) esetén jelölje $m(f)$ ($m(z)$) az f fa (a z gyökerű részfa) magasságát. Az S fa minden csúcsában feljegyezzük az itt gyökerező részfa szintjeinek számát. Ez a mező könnyen karbantartható a naiv beszúrás és törlés során. (Valójában mindössze 2 bit/csúcs extra információ is elég az *AVL*-tulajdonság helyreállításához. Ezt itt nem részletezzük.) A naiv beszúrás után a keresési út ismételt bejárásával megkapjuk a *legalsó* olyan csúcsot (ez legyen x), ahol az *AVL*-tulajdonság megsérül.

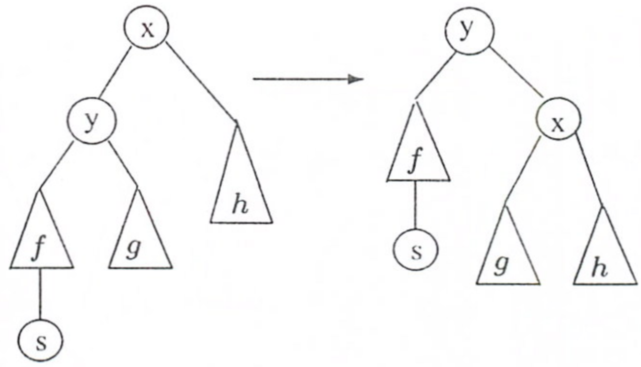


Az x definíciójából adódik, hogy x bal és jobb részfájának nem lehet ugyanaz a magassága. Az általánosság csorbítása nélkül feltehetjük, hogy a bal részfa magasabb, jelesül nem üres. (Ha a jobb részfa a magasabb, akkor az alább következő recept tükörképével, benne a *jobb* és a *bal* felcserélésével érhetünk célt.) Legyen ezek után a bal részfa gyökérpontja y, ennek a részfái pedig f (bal) és g (jobb). Az x jobb részfája legyen h. Legyen $m(y) = l$; ekkor szükségképpen $m(h) = l - 1$. Két esetet különböztetünk meg: a beszúrás során

(a) az s az f-be kerül;

(b) az s a g-be kerül.

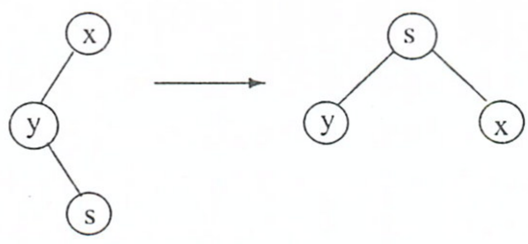
Nézzük először az (a) esetet. Ekkor $m(f) = m(g) = l - 1$. Ugyanis $m(f) < m(g)$ esetén a beszúrás nem tudná megsérteni az *AVL*-tulajdonságot x-ben. Másfelől $m(f) > m(g)$ azt jelentené, hogy y-ban is – tehát x alatt – sérül az *AVL*-tulajdonság. Innen adódik, hogy $m(f) = m(g) = m(y) - 1 = l - 1$. Eme ismeretek birtokában állítjuk, hogy az x-nél elvégzett jobb forgatás megoldja a problémát.



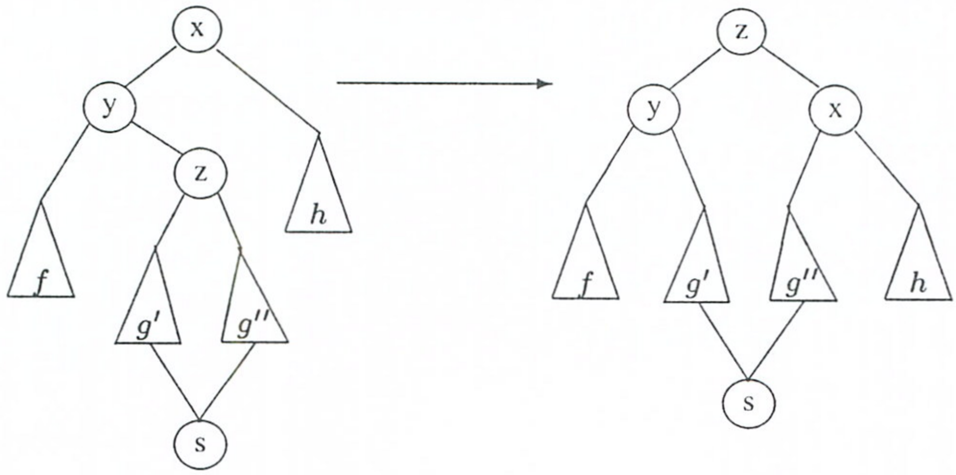
A forgatás után y mindkét részfájának a magassága l lesz, x új részfái g és h, mindkettő szintszáma $l-1$. Ezen csúcsok és részfák rendben vannak. Az x definíciója miatt f-ben az s beillesztése után sem sérülhet az *AVL*-tulajdonság. Végül megjegyezzük, hogy az új helyre került y feletti csúcsok magassága ugyanannyi marad a beszúrás és forgatás után, mint amennyi eredetileg volt; így az *AVL*-feltétel feljebb is megmarad a kereső út mentén.

Nézzük most a (b) lehetőséget, amikor is s az y jobb részfájába kerül. Ezt mindjárt két alesetre bontjuk.
(b1) az új csúcs y fia, azaz $l = 1$;
(b2) az új csúcs y-nak nem fia, más szóval $l > 1$.

A (b1) esetben y bal részfája és x jobb részfája is üres, ezért elegendő egy dupla forgatás x-nél.



A (b2) esetben a g részfa nem üres, hiszen $m(g) = l - 1 > 0$. Legyen a részfa gyökere z, ennek részfái g' és g''. Az s kulcs e két részfa valamelyikébe kerül, a továbbiak szempontjából közömbös, hogy melyikbe. Ekkor $m(f) = l - 1$ (mert y-ban az *AVL*-feltétel teljesül), és $m(g') = m(g'') = l - 2$ (mert z-ben sincs baj az *AVL*-tulajdonsággal). Egy kettős forgatás mindent helyretesz.



A részfa új gyökere z kiegyensúlyozott lesz, $m(z) = l + 1$. Ennyi volt $m(x)$ a beszúrás előtt. A z bal fia y balsúlyos vagy kiegyensúlyozott lesz aszerint, hogy s a g' vagy g'' részfába kerül-e. Ugyanígy z jobb fia x lesz, ami jobbsúlyos vagy kiegyensúlyozott. Ezzel az algoritmus ismertetését befejeztük. Az itt vázolt eljárás költsége nyilvánvalóan arányos a naiv beszúrás költségével; az $m(S) \leq 1.44 \log_2(n+1)$ egyenlőtlenségből látszik, hogy ez $O(\log n)$. □

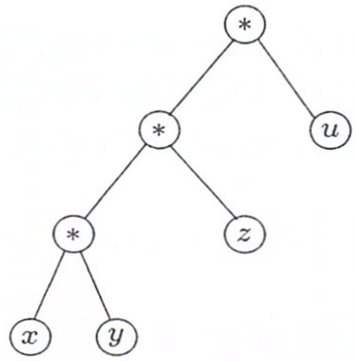
A törlés algoritmusa is hasonló felépítésű. Először a naiv módszerrel eltávolítjuk a törölni kívánt elemet, majd – ha szükséges – alkalmas forgatásokkal helyreállítjuk az *AVL*-tulajdonságot.

Tétel: *Az n szögpontú AVL-fából való naiv törlés után legfeljebb $1.44 \log_2 n$ (szimpla vagy dupla) forgatás helyreállítja az AVL-tulajdonságot.*

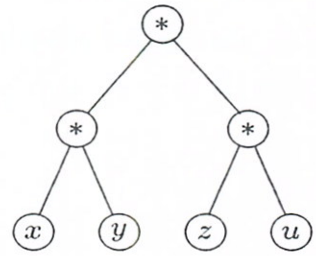
Ezt nem bizonyítjuk. A helyreállító fázis valamivel bonyolultabb, mint a beszúrásnál alkalmazott módszer. Előfordulhat, hogy a törlési út mentén több csúcsnál is kell forgatást végezni. A törlés költsége evvel együtt is $O(\log n)$.

Feladat: Adjunk példát egy olyan *AVL*-fára, melynél egy alkalmas törlés után nem állítható helyre az *AVL*-tulajdonság egyetlen (szimpla vagy dupla) forgatással.

Megjegyzés: A forgatások szoros kapcsolatban vannak az *asszociatív szabállyal.* Képzeljük el, hogy van egy kétváltozós műveletünk, mondjuk $*$. E művelet segítségével kifejezéseket készíthetünk (pl. $(x * y) * (z * u)$, ahol x, y, z, u változók). Egy kifejezésnek természetes módon megfelel egy bináris fa, melynek levelei a változók. Ebben a kifejezésfában forgatások felelnek meg a kifejezés asszociatív szabály szerinti átalakításainak.



$((x * y) * z) * u$



$(x * y) * (z * u)$

Az ábrán baloldalt levő kifejezésfából a gyökérnél elvégzett jobbforgatással keletkezett a másik fa. Látható, hogy az új kifejezés a régiből az asszociatív szabály egyszeri alkalmazásával származtatható.

3.6. További megjegyzések kiegyensúlyozott fákról

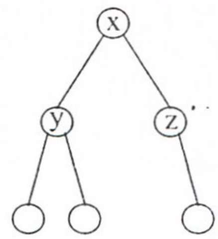
Az *AVL*-tulajdonság olyan szabályt, előírást jelent, melyet ha megtartunk, akkor kiegyensúlyozott fát kapunk. Az AVL-fák magassága legfeljebb mintegy 1.44-szerese az ideális fákénak. Ez biztosítja, hogy a bennük való keresés hatékony. Az *AVL*-tulajdonság csak egy a lehetséges kiegyensúlyozottsági feltételek közül. Más olyan tulajdonságok is vannak, amelyek hasonló módon szabályozzák a fa alakját, és a megőrzésük sem jelent túl nagy algoritmikus nehézséget. E tulajdonságok (konstrukciók) közül kettőt érintünk a továbbiakban.

HB[k]-fák (C. C. Foster, 1973)
Legyen $k \geq 1$ egy egész szám. Egy bináris keresőfa $HB[k]$-fa, ha minden x csúcsára teljesül, hogy $|m(bal(x)) - m(jobb(x))| \leq k$. Az előírás az *AVL*-feltétel általánosítása. Speciálisan a $HB[1]$-fák éppen az *AVL*-fák. Az *AVL*-fák esetéhez hasonló, bár valamivel bonyolultabb módon megmutatható, hogy a $HB[k]$-feltétel általában is kiegyensúlyozott fákat eredményez.

Feladat: Az előbbi definícióból kizártuk a $k = 0$ esetet. Mi lehet ennek a magyarázata? Miért alkalmatlanok a $HB[0]$-fák a keresőfa adatszerkezet megvalósítására?

Súlyra kiegyensúlyozott fák (J. Nievergelt, B. Reingold, C. Wong, 70-es évek)
Az előző feltétel a részfák magasságának a szabályozásával biztosította a kellően telt alakú fákat. Erre a célra más kombinatorikus tulajdonságok is alkalmasak. Ilyen jellemző lehet például a részfák csúcsainak száma.

Legyen x egy bináris fa csúcsa. Az x csúcs *súlya* az x-gyökerű részfában levő csúcsok száma. Az x csúcs súlyának a jele $s(x)$. Például a következő fában $s(x) = 6$, $s(y) = 3$ és $s(z) = 2$.



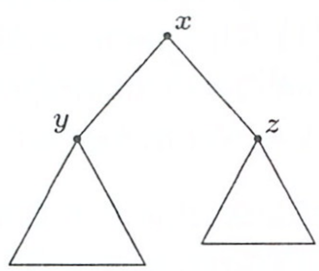
Definíció: *Egy bináris keresőfát súlyra kiegyensúlyozott fának (röviden SK-fának) nevezünk, ha minden x belső csúcsára teljesül, hogy*

$$\sqrt{2} - 1 < \frac{s(bal(x))}{s(jobb(x))} < \sqrt{2} + 1.$$

Láthatjuk, hogy az SK-feltétel jobb-bal szimmetrikus. Az egyenlőtlenségek reciprokát véve kiderül, hogy $\sqrt{2} - 1 < \frac{s(jobb(x))}{s(bal(x))} < \sqrt{2} + 1$ is igaz.

Feladat: Igazoljuk, hogy a leheletnyivel szigorúbb $1/2 < \frac{s(bal(x))}{s(jobb(x))} < 2$ korlátokat már csak az l szintből álló, $2^l - 1$ pontú bináris fák a teljesítik.

Most megmutatjuk, hogy az SK-feltétel tényleg kiegyensúlyozott fákhoz vezet; a magasságkorlátban szereplő c állandó 2-nek adódik. Legyen evégből S egy SK-fa, melyre $s(S) = n$ és $m(S) = k$. Legyen x az S egy belső csúcsa, melynek bal fia y, a jobb fia pedig z.



Ekkor $s(x) > s(y) + s(z) > (\sqrt{2} - 1)s(z) + s(z) = \sqrt{2}s(z)$. Az első egyenlőtlenség a súly definíciója, a második pedig az SK-feltétel miatt igaz. Innen az SK-feltétel jobb-bal szimmetriáját használva megállapíthatjuk, hogy érvényes az $s(x) > \sqrt{2}s(y)$ egyenlőtlenség is. Legyenek most $x_1, x_2, \ldots, x_k$

egy k-hosszúságú gyökértől levélig menő út csúcsai. Az S gyökere x_1, amiből $n = s(S) = s(x_1)$, továbbá $x_1, x_2, \ldots, x_{k-1}$ belső csúcsai S-nek. Az előbb nyert egyenlőtlenségek ismételt alkalmazásával kapjuk, hogy

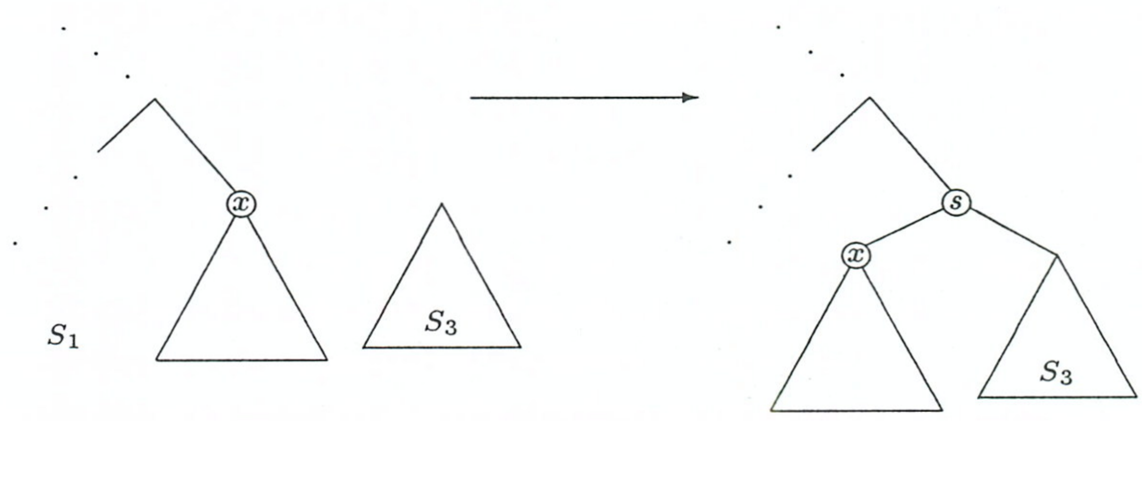
$$n = s(x_1) > \sqrt{2}s(x_2) > (\sqrt{2})^2 s(x_3) > \cdots > (\sqrt{2})^{k-1} s(x_k) = (\sqrt{2})^{k-1}.$$

A lánc elejét a végével összevetve $n > (\sqrt{2})^{k-1}$, amiből logaritmusra térve $2\log_2 n + 1 > k$ adódik. Egy n csúcsból álló SK-fa magasságának maximuma legfeljebb $2\log_2 n + 1$.

Egy további alkalmazás

A kiegyensúlyozott fák alkalmasak a tárgyalt alapfeladatokon kívül más problémák hatékony megoldására is. Példaként említjük a rendezett listák összefűzésének feladatát. Az elemek inorder sorrendjére gondolva egy *AVL*-fa felfogható rendezett listának. Tegyük fel, hogy van két rendezett listánk, S_1 és S_2, melyeket egy-egy *AVL*-fában tárolunk. Tegyük fel továbbá, hogy az S_2-ben szereplő kulcsok mind nagyobbak az S_1 kulcsainál. Célunk a két lista egyesítése; a két *AVL*-fát szeretnénk minél hatékonyabban egyetlen *AVL*-fává összefűzni. Megmutatjuk, hogy ez megtehető $O(\log n)$ költséggel, ahol n a két lista összesített elemszáma. A fákról feltételezzük, hogy a csúcsai tartalmazzák az ott gyökerező részfák magasságát. Legyen $m(S_1) \geq m(S_2)$. (A fordított eset hasonlóan kezelhető.) Az eljárás fő lépései a következők:

1. Megkeressük és töröljük S_2 legkisebb elemét (legyen ez s). A törlés eredményeként kapott fát jelölje S_3.
2. S_1 gyökerétől indulva, és rendületlenül jobbfelé haladva megkeressük az első olyan x csúcsot, melyre $m(x) - m(S_3) = 0$ vagy 1. (A feltétel azért tartalmaz két lehetőséget, mert egy lépésnél a magasság 1-gyel vagy 2-vel csökkenhet.)
3. Ezután x helyére egy új csúcsot teszünk, melynek kulcsa s. Az új csúcs jobb részfája S_3, bal részfája pedig az S_1 fa x-gyökerű részfája lesz.



Könnyen ellenőrizhető, hogy az így kapott fa teljesíti a keresőfa-tulajdonságot és az *AVL*-feltételt is. A költségkorlát azonnal következik abból, hogy S_1 és S_2 magassága is $O(\log n)$.

3.7. Egy önszervező megoldás: az S-fák

Kvasztics felriadt álmából, meglátta az ájult őrt, és fáradtan körülnézett.
– Csak menjetek nyugodtan... – mondta – az őrt megvizsgálom és ellátom kötéssel.
– Az nem elég – súgta Tuskó – a száját is tömje be, ha már ápolás alá veszi.
REJTŐ JENŐ: A három testőr Afrikában

Az eddig vett hatékony keresőfák algoritmusai zord szigorúsággal vigyáznak a fák alakjára. Ezáltal a műveletek legkedvezőtlenebb eseteire is biztosítani tudják a $O(\log n)$ nagyságrendű korlátot, ahol n a tárolt elemek száma. Egészen másféle szemléletet képvisel az itt bemutatásra kerülő önszervező bináris keresőfa-konstrukció, melynek neve S-fa. Az elnevezés az angol *splay tree* (kifordított fa) kezdőbetűjéből származik. Az S-fákat D. D. Sleator és R. E. Tarjan javasolták 1983-ban.

Egy S-fa a tárolási szerkezetét illetően egyszerűen egy bináris keresőfa. Érdekessé és hatékonnyá az alapműveletekre javasolt módszerek teszik. Ezek az algoritmusok semmiféle közvetlen figyelmet nem szentelnek a fa alakjának. Előfordulhat például, hogy a fában hosszú utak vannak, és ezért bizonyos keresések lassúak. Az S-fák tehát nem feltétlenül kiegyensúlyozottak. Ezzel szemben hosszabb operációsor (keresések, beszúrások, törlések, stb.) alatt „tanulnak": az egyes elemek elérési gyakoriságai és ezen elérések időbeli eloszlása szerint változtatják alakjukat. Úgy is mondhatjuk, hogy a fa idomul a felhasználói igényekhez.

Az S-fák algoritmusai mögött egy nagyerejű és sok esetben használható gondolat húzódik meg. Ha a felhasználói igények teljesítése (mondjuk keresés) közben eljutunk valahova, akkor nem távozunk onnan szűkkeblűen, pusztán csak a kötelezőket elvégezve. Rászánunk még valamennyi időt, hogy szépítsük, alakítsuk a környéket ahová kerültünk. Ez az idő (nagyságrendileg) nem több, mint amit a felhasználói kérésre amúgy is fordítanunk kell. A ráfordítás javítja a szerkezet hatékonyságát, aminek az eredménye az lesz, hogy a jövőbeli kéréseket gyorsabban tudjuk teljesíteni. Ezzel a filozófiával később is találkozni fogunk a szekvenciális keresés önszervező módszereinél és az UNIÓ-HOLVAN adatszerkezetnél.

Az S-fák alkalmazkodó képességének két konkrétabb megnyilvánulását említjük:

1. Egy hosszú műveletsor esetén az egy műveletre eső költség konstans szorzó erejéig optimális lesz. Ha tehát a fa elég sokáig él, akkor az összesített időigénye nem lesz számottevően rosszabb a legjobb kiegyensúlyozott fákénál. E tulajdonság pontosabb megfogalmazása a szakasz végén található tétel.

2. A kiegyensúlyozott fáknál jobb teljesítményt nyújtanak olyan alkalmazási helyzetekben, ahol az egyes elemekkel kapcsolatos elérési igények „időben csomósodnak". Az ilyen csomós helyzetre suta, de szemléletes példa egy kórház betegnyilvántartása. (A sutaság abból ered, hogy egy kórházi nyilvántartást külső táron érdemes kezelni, míg a bináris keresőfák belső memóriás szerkezetek.) Ha valaki bekerül a kórházba, akkor a róla szóló rekord igen aktív lesz; viszonylag gyors egymásutánban kerülnek bele a különféle vizsgálatokkal, kezelésekkel kapcsolatos feljegyzések. Ha viszont az illető éppen nem beteg, akkor meglehetősen ritkán van szükség erre a rekordra.

Az S-fák ezen alkalmazkodási képességét egy igen egyszerű, ugyanakkor csillogóan elegáns visszacsatolási mechanizmus biztosítja. Ennek lényege, hogy ha a fában tárolt $s \in U$ elemmel kapcsolatos elérési igény érkezik (pl. KERES(s, f)), akkor az igényt kezelő algoritmus ezt úgy „értelmezi", hogy az s fontossága nőtt. Az s elemet alkalmas forgatásokkal a fa gyökerébe mozgatja. Ezen felül az s-hez vezető kereső út mentén levő elemek is valamivel fontosabbak lettek; a módszer őket is közelíti a gyökérhez. Ennek eredményeként az adott időszakban gyakran használt elemek közel lesznek a fa tetejéhez, a kevésbé aktívak pedig lassan a levelek felé vándorolnak. A gyakran használt elemek ezért nagyon gyorsan elérhetők. A kórházi példánál maradva a fa csaknem olyan teljesítményt nyújt, mintha csupán az éppen kezelt betegek rekordjaiból állna a nyilvántartás.

Az S-fák műveleteinek pontos leírásához bevezetünk néhány jelölést. Legyenek f, f' S-fák, x, y, z kulcsok, az U rendezett univerzum elemei. Utóbbiakat azonosítani fogjuk az őket tartalmazó fabeli csúcsokkal. Ez nem fog félreértést okozni.

A keresőfákra jellemző KERES(x, f), BESZÚR(x, f) és TÖRÖL(x, f) műveleteket a szokásos módon értelmezzük. A RAGASZT(f, f') művelet az f és f' S-fákból egyetlen S-fát szervez, feltéve, hogy $x < y$ teljesül minden $x \in f$ és $y \in f'$ kulcsra. A VÁG(x, f) művelet szétvágja f-et az f' és f'' S-fákra úgy, hogy $y \leq x \leq z$ teljesül minden $y \in f'$ és $z \in f''$ csúcsra.

A korábban említett önszervező-szépítő képesség a KIFORDÍT elnevezésű eljárásba van építve. Ennek a definíciója az erejéhez képest meglepően egyszerű.

KIFORDÍT(x, f) átszervezi az f S-fát úgy, hogy x lesz az új gyökér, ha $x \in f$; különben f gyökere x valamelyik szomszédja lesz:
vagy $\max\{y \in f;\ y < x\}$ vagy $\min\{y \in f;\ y > x\}$.

A KIFORDÍT művelet az egész módszercsalád lelke. Segítségével a többi eljárás könnyen megvalósítható:

Állítás: *Az ismertetett műveletek mindegyike megvalósítható konstans számú KIFORDÍT és konstans számú elemi operáció (összehasonlítás, mutató beállítás, stb.) segítségével.*

Bizonyítás: Csak két művelet, a RAGASZT és a TÖRÖL esetét nézzük meg közelebbről. A többi módszer összerakását feladatként az olvasóra hagyjuk.

RAGASZT(f, f') végrehajtását a KIFORDÍT($+\infty, f$) hívással kezdjük. Itt $+\infty \in U$ egy az f minden eleménél nagyobb kulcsot jelent. Legyen x az eredményül kapott f^* fa gyökere. A KIFORDÍT specifikációja szerint az x az f^* legnagyobb kulcsa. Ebből következik, hogy x-nek nincs jobboldali fia. Legyen tehát az x jobboldali fia az f' fa gyökere. Az így kapott fára teljesül a keresőfa-tulajdonság, mert a hívási feltétel szerint f' kulcsai nagyobbak x-nél.

TÖRÖL(x, f) első lépése KIFORDÍT(x, f). Ha az ekkor kapott fa gyökere nem x, akkor készen vagyunk; ez azt jelenti, hogy $x \notin f$. Feltehetjük ezután, hogy a fa gyökere x. Legyenek f_1 és f_2 az x gyökér részfái. A törlést RAGASZT(f_1, f_2) végrehajtásával fejezhetjük be. □

KIFORDÍT(x, f) implementációja

Hasznos lesz a következő jelölés: legyen x az f fa egy csúcsa, melynek apja y; ez esetben FORGAT(x) jelölje azt az egyszeres forgatást, mely x-et y apjává teszi.

Először a bináris keresőfákban szokásos módszerrel megpróbáljuk megtalálni x-et f-ben. Ez vagy sikerül, vagy pedig ha $x \notin f$, akkor az x-nek a rendezés szerinti egyik szomszédjánál ($\max\{y \in f;\ y < x\}$ vagy $\min\{y \in f;\ y > x\}$) végzünk. Mindegyik esetben azt az elemet kell majd felvinnünk a fa tetejére, amelyet a keresés során utoljára elértünk. Feltehetjük ezért, hogy $x \in f$, és már meg is leltük x-et f-ben.

Ezután a kezünkben levő x elemet az alább következő recept ismételt alkalmazásával felvisszük a fa tetejére. Az eljárásdarab egy végrehajtása maximum két szinttel viszi feljebb x-et. Egy menetben a 0-3. lépések közül pontosan egy hajtódik végre. A döntéshez szükséges ellenőrzés könnyű, mert az x-hez vezető kereső út utolsó néhány (legfeljebb három) csúcsának és a köztük menő éleknek az ismeretében elvégezhető.

(0) Ha x gyökér, akkor készen vagyunk. (* A továbbiakban jelölje y az x apját. *) (1) Ha x-nek nincs nagyapja, akkor FORGAT(x), különben (2) ha x és y is baloldali (jobboldali) gyerek, akkor FORGAT(y), majd FORGAT(x), különben (3) ha x és y különböző oldali gyerekek, akkor FORGAT(x), majd ismét FORGAT(x).

KIFORDÍT(x, f) fontos járulékos hatása, hogy az x kereső útján levő csúcsok közelebb kerülnek a fa tetejéhez. A keresőfa-műveletek – a bennük levő KIFORDÍT-hívásnak köszönhetően – változtatják a fa alakját. Ez a változtatás annál több csúcsot érint, minél mélyebben levő elemre alkalmazzuk a kifordító eljárást.

A következő eredmény a korábban említett 1. tulajdonság pontosabb megfogalmazása. Lényegében azt mondja, hogy egy hosszú műveletsor összköltsége állandó szorzó erejéig a kiegyensúlyozott fákra jellemző korláton belül marad: az egy műveletre eső átlagos költség $O(\log n)$, ahol n a fa csúcsainak a száma. A bizonyítást mellőzzük.

Tétel: *Egy üres S-fából induló olyan m műveletből álló sorozat költsége, melyben n beszúrás van,* $O(m \log n)$. □

3.8. Szófák

Az eddigi keresőfa-konstrukciók csak annyit tételeztek fel az U kulcshalmazról, hogy az rendezett halmaz. A rendező módszereknél a kulcsokról való finomabb ismeretek hatékonyabb algoritmusokhoz vezettek; gondoljunk csak a láda- vagy a radixrendezésre. Némiképp hasonló a helyzet a keresőfa-műveleteknél. Bizonyos speciális szerkezetű kulcsokra érdekes *ad hoc* megoldások adhatók. Szép és hasznos példaként említhetők a *szófák*. A szerkezet angol neve egy fantáziaszó: *trie*, amely a retrieval szó közepéből való négy betűre szándékozik utalni (kiejtése viszont a *try* mintáját követi).

Legyen Σ egy véges halmaz, amit úgy tekintünk, mint egy nyelv betűkészletét (*abc*-jét). Jelölje Σ^* a Σ-beli elemekből alkotott véges hosszú sorozatok, azaz szavak halmazát. Itt most feltesszük, hogy adott egy rendezés (amit *abc*-sorrendnek tekintünk) a Σ halmazon. Ebből azonnal adódik egy rendezés Σ^*-on, nevezetesen a betűrenden alapuló lexikografikus rendezés. A szófák hatékony keresőfa-

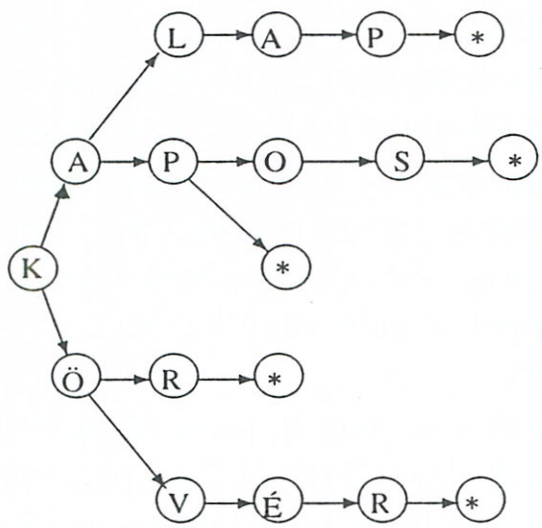
konstrukciót adnak arra az esetre, amikor $U = \Sigma^*$, a rendezés pedig a lexikografikus. A szófák tehát akkor alkalmazhatók, ha a kulcsok szavak.

A szófát gyökeres, a gyökértől távolodóan irányított fának képzelhetjük el. A fa csúcsai lényegében mutató típusú tömbök, $\Sigma \cup \{*\}$ elemeivel indexelve. Itt $* \notin \Sigma$ egy speciális végjel, amiről feltesszük, hogy a betűrend szerint Σ minden betűje megelőzi. Úgy is fogalmazhatunk, hogy a tömbök utolsó, $|\Sigma|+1$. elemének az indexe $*$. A tömbök elemeiben levő mutatók adják a fa éleit. Egy csúcsnak ezek szerint maximum $| \Sigma | +1$ fia lehet. Egy tömbelem vagy NULL (jelezve hogy üres, tehát nem tartalmaz igazi mutatót) vagy egy mutató egy fiúra (azaz egy másik tömbre) vagy a $*$ végjel. A fában tárolt szavakat gyökértől levélig, azaz a $*$ végjelig menő utak jelentik.

Például ha Σ a magyar *abc*, és a KAP szót szeretnénk tárolni, akkor a gyökeret jelentő tömb K indexű eleméből mutató mutat egy másik tömbre, ennek A indexű elemében mutató van egy további x tömbre, melyre $x[\text{P}] = *$. Bonyolultabb a helyzet, ha a KAPU szó is szerepel a szófánkban. Ekkor $x[\text{P}]$ valódi mutató, mondjuk az y tömbre. Azt a tényt pedig, hogy a KAP szó is szerepel, úgy jelezhetjük, hogy $*$-ot írunk $y[*]$-ba. Másképpen fogalmazva: a problémát, hogy egyik szó prefixe a másiknak – mint a KAP a KAPU-nak – úgy kezeljük, hogy minden szó végére egy $*$-ot képzelünk, és $*$ csak a szó végén fordulhat elő. Az $x[\text{P}]$-ben gyökerező részfa azoknak a tárolt szavaknak felel meg, amelyeknek az első három betűje KAP.

A tömbök helyett lineáris listák is használhatók. Ezáltal megtakaríthatjuk a NULL értékű tömbelemek helyét. Ennél a megközelítésnél viszont némi gondot okoz a listák hatékony kezelése.

Legyen Σ továbbra is a magyar *abc*. A KÖR, KÖVÉR, KAPOS, KAP, KALAP szavakat tartalmazó fa listás ábrázolása így képzelhető el:



Itt egy adott csúcs fiai tekinthetők egy listán levőknek. Így például a gyökér egy

egyelemű lista, a második szinten levő A betű fiai, L és P szintén egy (kételemű) listát képeznek.

A szófa adatszerkezet érzéketlen a beszúrások sorrendjére, a fa alakja csak a tárolt szavak összességétől függ. A tömbbel megvalósított szófában való keresés igen hatékony. Ugyanis a bemenő szó hosszához mérten nem kell többet lépnünk a mutatók mentén, mint amennyi a szóban a betűk száma. Megmutatható, hogy m betűs *abc* feletti n véletlen kulcsból álló szófa esetén a keresés átlagosan mintegy $\log_m n + c$ mutató követésével befejeződik, ahol c az m-től és n-től is független állandó.

4.

Hash-elés és szekvenciális keresés

Hány hajó ütközött itt össze; jóllehet voltak jelzőfényeik, kürtjeik és vészharangjaik! JULES VERNE
(A *Nemo kapitány* elbeszélőjének tűnődése Új-Fundland vizein.)

Ebben a fejezetben először egy olyan módszercsaládot ismertetünk, amely a keresés, beszúrás, törlés és módosítás gyors és egyszerű megvalósítását teszi lehetővé. A keresőfákhoz képest fontos eltérés, hogy nem tételezzük fel a lehetséges kulcsok összességének (az U univerzumnak) a rendezettségét. Ennek megfelelően itt nem lesznek rendezéshez kapcsolódó elérési igények, mint pl. a MIN. A cél egy $S \subseteq U$ kulcshalmazzal azonosított állomány megszervezése úgy, hogy a fenti műveletek hatékonyak legyenek. A megoldások, amelyekkel megismerkedünk, *átlagos értelemben* igen gyorsak. A legrosszabb esetekben viszont nagyon lassúak is lehetnek. Olyan alkalmazásoknál jönnek tehát szóba, ahol az átlagosan jó teljesítmény az igazán fontos, és a ritkán előforduló rosszabb válaszidők nem okoznak gondot.

Először egy rövid példával szeretnénk szemléltetni a hash-elés hátterét, alapötletét és a felmerülő problémákat. Tegyük fel, hogy magyar állampolgárok adatait szeretnénk tárolni. Az állomány egy rekordjában olyan adatok (mezők) szerepelhetnek, mint például név, lakcím, személyi szám, stb. A rekordokat azonosító kulcsként a személyi szám szolgálhat. A személyi szám 11 jegyű, ebből egy redundáns, és a további ismert korlátozásokat is figyelembe véve (a hónapok száma, a hónapok napjainak száma, stb.) mintegy $2 \cdot 10^2 \cdot 12 \cdot 31 \cdot 10^3 \approx 74$ millió kulcs lehetséges. Ugyanakkor még jó adag demográfiai optimizmus esetén sem kell több, mint 11 millió tényleges rekorddal számolnunk a közeljövőben. Nagyvonalúan – ami mint később látni fogjuk, hasznos ebben az összefüggésben is – gondolkodhatunk úgy, hogy 12 millió rekord számára foglalunk helyet. Legyen a rekordok

(logikai) címeinek tartománya a $[0, 12 \cdot 10^6 - 1]$ intervallum egészeinek I halmaza. Ezután a feladatot felfoghatjuk úgy is, hogy egy *olyan megfeleltetést keresünk, mely minden kulcshoz egy címet rendel.* Olyan h függvényre gondolunk, amelynek az értelmezési tartománya a lehetséges kulcsok halmaza, értékkészlete pedig I, a logikai címek tartománya. Kényelmes volna, ha $K \neq K'$ esetén $h(K) \neq h(K')$ teljesülne. Ez azonban képtelenség, hiszen az értelmezési tartomány jóval nagyobb, mint az értékkészlet, ezért h nem lehet injektív. Az ilyen *ütközések* tehát elkerülhetetlenek. Sőt, bizonyos értelemben elég gyakran előfordulnak. Erre világít rá a születésnap-paradoxon (szokás még von Mises-paradoxonnak is nevezni): egy legalább 23 tagú társaságban legalább $1/2$ valószínűséggel van két személy, akiknek megegyezik a születésnapjuk. Általában megmutatható, hogy ha a h függvény értékkészlete M elemű, akkor $\sqrt{2 \log 2 \cdot M}$ véletlenül választott argumentum között legalább $1/2$ valószínűséggel lesz kettő, melyre h ugyanazt az értéket adja. E tény azt sugallja, hogy jó esély van ütközésre még akkor is, amikor címtartomány méretéhez képest viszonylag kevés rekordunk van a tárolt állományban.

A *hash-elés* vagy *hash-kódolás* módszerét használó tárolási technikák alapvető közös vonása, hogy egy alkalmas h függvény, az úgynevezett hash-függvény segítségével kísérlik meg kiszámítani a beillesztendő rekord (logikai) címét. A h függvény a K kulcshoz a $h(K)$ címet rendeli. A módszerek első közelítésben a K kulcsú rekordot a $h(K)$ címnek megfelelő helyre próbálják tenni. Ez nem lehetséges ütközés esetén: amikor egy másik K' rekord érkezik, melyre $h(K) = h(K')$. A teendők fontos része éppen ezért az *ütközések feloldása.* Olyan megoldásokról van itt szó, amelyek ütközéskor helyet találnak a később jött rekordoknak. Az egyes módszereket az ütközések kezelése alapján szokás osztályozni. Ezt tesszük mi is.

A másik alapvető kérdést a *megfelelő hash-függvény* találása, kiválasztása jelenti. Olyan könnyen számítható h függvényeket keresünk, amelyek az alkalmazás során felmerülő kulcshalmazokon minél kevesebb ütközést okoznak. Ennek a mikéntjeivel is foglalkozunk.

A *szekvenciális kereséssel* már találkoztunk a rendezések tárgyalásakor. Ott rendezett állományokban való keresésre használtuk a módszert. A fejezet végén ejtünk néhány szót arról az esetről, amikor az állomány nem feltétlenül rendezett.

4.1. A hash-elés alapjai

Két igen szerteágazó módszer legegyszerűbb változatait ismertetjük. Ezek a *vödrös hash-elés* és a *nyitott címzés.* Az első technikát, a vödrös hash-elést elsősorban külső táron levő nagyméretű állományok kezelésére használják. Szinte minden komoly adatbáziskezelő rendszer ad ilyen jellegű tárolási lehetőséget. A nyitott cím-

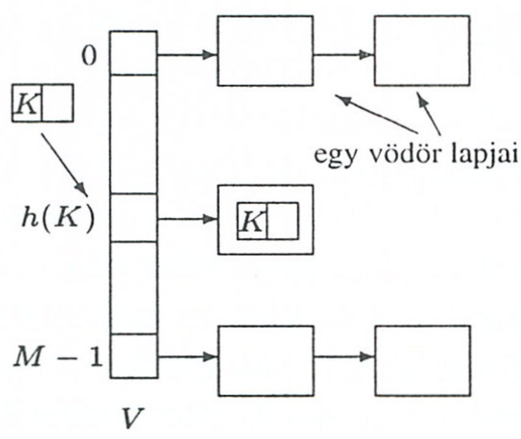
zésű módszerek ezzel szemben főleg a belső memóriában levő állományok gyors kezelését hivatottak biztosítani. Jellegzetes alkalmazásként említhetjük a programok fordításakor keletkező szimbólumtáblák kezelését vagy a később bemutatandó Lempel–Ziv–Welch-módszer szótárának a tárolását.

A továbbiakban mindkét esetre érvényesen feltesszük, hogy van egy h hash-függvényünk, amely a K kulcshoz a $h(K)$ logikai címet rendeli. A $h(K)$ cím egy egész a $[0, M-1]$ intervallumból. Azért beszélünk *logikai címekről*, mert a $h(K)$ értékek többnyire nem jelentenek tényleges fizikai helymegjelöléseket. Ugyanakkor feltehetjük, hogy a rendszer biztosít egy mechanizmust, amely a $[0, M-1]$ intervallum egészeit a tényleges adatmozgatást végző réteg számára értelmezhető címekké fordítja.

4.1.1. Vödrös hash-elés

A módszert szokásos *láncolásnak* is nevezni. Lényeges eleme a $V[0 : M-1]$ vödörkatalógus. Amint a jelölés is sugallja, V-t hasznos egy a h értékkészletének elemeivel indexelt tömbnek elképzelni. A kép nem teljesen hűséges, mert V nagy is lehet. Előfordulhat, hogy részben vagy egészében háttértáron kell elhelyezni. A V elemei mutatók, pontosabban lapláncok fejei. A lapláncokat szokás *vödröknek* nevezni. A vödrök lapjain helyezkednek el a tárolni kívánt rekordok. Nevezetesen a $V[i]$ mutatóval kezdődő vödörbe kerülnek azok a rekordok, amelyek K kulcsaira $h(K) = i$ teljesül.

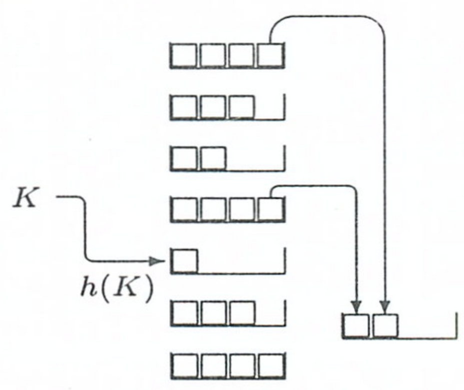
Vegyük szemügyre ezután a szerkezet algoritmusait! Tegyük fel, hogy a K kulcsú rekordot akarjuk beilleszteni. Először kiszámítjuk a $h(K)$ értéket, majd megnézzük a vödörkatalógus $V[h(K)]$ bejegyzését. Ez egy mutató arra a vödörre (lapláncra), melyben a $h(K)$ hash-értékű rekordokat kívánjuk tárolni. A vödör általában egy kicsi, legfeljebb néhány lapból álló állomány. Ebben elhelyezzük a tárolni kívánt rekordot. A következő ábrán a V vödörkatalógust függőleges helyzetben mutatjuk; a vödrök a lapokból álló vízszintes sorok.



A rekordnak a vödörben való elhelyezése több módon is történhet. Gyakran használatos és egyszerű megoldás, hogy a rekordot az első szabad helyre tesszük. Ha szükséges, új lap hozzáfűzésével bővítjük a láncot. Másik szokásos eljárás, hogy a rekordokat kulcs szerint rendezve tartjuk a vödörben. Ekkor – a beszúrásos rendezésnél tapasztaltakhoz hasonlóan – előfordulhat, hogy több rekordot is mozgatni kell a vödrön belül.

A keresés módszere ezek után nyilvánvaló. A K kulcsú rekord keresése avval kezdődik, hogy kiszámítjuk a $h(K)$ értéket. Ha a rekord a tárolt állományunkban van, akkor csak a $V[h(K)]$ mutatótól induló vödörben lehet. Ezután tehát a vödörkatalógus $V[h(K)]$ mutatóját követve a vödör első lapjára lépünk. A vödörben szekvenciális kereséssel dolgozhatunk. Addig megyünk a lapláncmentén, amíg vagy megtaláljuk a rekordot, vagy pedig kiderül, hogy az nincs a tárolt állományban. A törléssel és a módosítással kapcsolatos teendők a keresés után már kézenfekvőek, így nem részletezzük őket. Csak annyit jegyzünk meg, hogy ha a módosítás a kulcsot is érinti, akkor törlés, majd újbóli beillesztés válhat szükségessé, hiszen a kulcs módosulásával annak h-értéke is változhat.

A vödrös hash-elés kitűnően alkalmazható külső tárakon elhelyezkedő állományok kezelésére. A módszer ötlete jól illeszthető a használt fizikai tárolási médium sajátosságaihoz. A vödrök gyakran megvalósíthatók a lapláncnál alacsonyabb, hatékonyabb szinten. Egy vödör lehet pl. a mágneslemez egy sávja (track) vagy annak írás/olvasás szempontjából folytonos része. Ilyenkor az utolsó sáv szolgálhat a túlcsordulások kezelésére:



Mi mondható a módszer költségéről? Mivel külső táras adatszerkezetről van szó, a lapelérések számát kell vizsgálnunk. Ezt szem előtt tartva világos, hogy a láncolási módszer időigényét (költségét) a lapláncok hossza határozza meg. Tegyük fel, hogy M vödör van, és l-lapnyi rekordot tárolunk. Ekkor egy vödörbe átlagosan $\approx l/M$ lap kerül. Ennyi lesz tehát az átlagos lánchossz. Ha a keresés során minden vödörhöz ugyanakkora eséllyel fordulunk, akkor a keresés átlagos költsége legfeljebb $1 + l/M$ lapelérés. Itt azzal a feltételezéssel éltünk, hogy a

vödörkatalógus is háttértáron helyezkedik el, és $V[h(K)]$ olvasása egy lapelérésbe kerül. Az átlagszámításban hallgatólagosan feltettük még azt is, hogy a h függvény elég egyenletesen szórja szét az érkező kulcsokat a $[0, M-1]$ intervallumban.

A módszer alkalmazása előtt fontos méretezési feladat a katalógus elemszámának, az M paraméternek a meghatározása. Általában ismerjük (esetleg csak közelítőleg) a tárolni kívánt állomány nagyságát kifejező l értéket. Ennek birtokában az M értékét célszerű úgy választani, hogy a várható l/M hányados közel legyen 1-hez. Nem szokatlan a 20 százalékos rátartás, vagyis többlethely lefoglalása sem.

Példa: Tegyük fel, hogy egy maximum 1 000 000 rekordból álló állományt szeretnénk láncolásos módszerrel kezelni. Tegyük fel, hogy egy lapon 5 rekord fér el. Ekkor $l = 1\ 000\ 000/5 = 200\ 000$. Az M paraméter értékét 220 000 – 240 000 körülinek érdemes választani. Ha a katalógus elemei is diszken vannak, és egyenként elérhetők 1 lépésben, akkor a keresés átlagos költsége valamivel 2 lapelérés alatt marad.

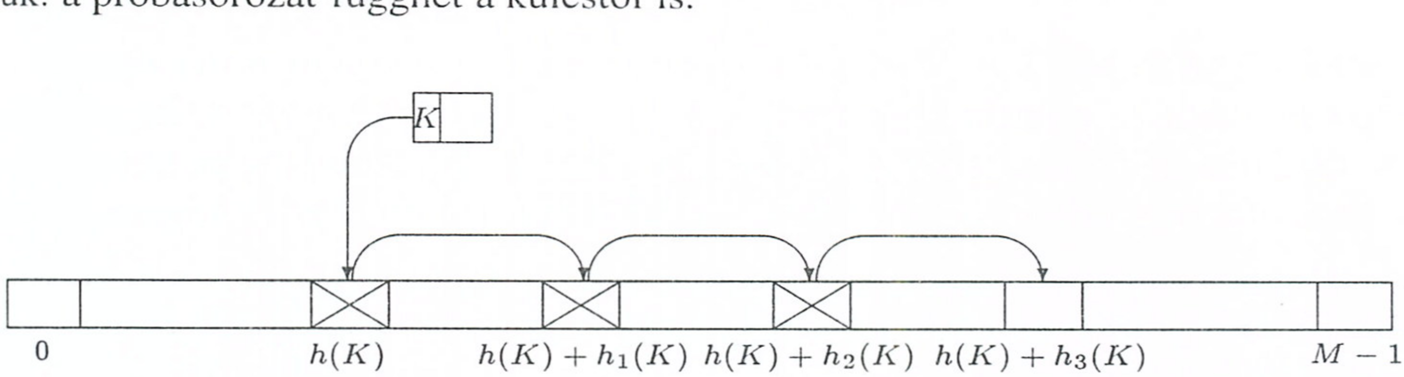
Végezetül megemlítjük, hogy a láncolás módszere belső memóriás megoldásként is versenyképes. Az előbbiekhez viszonyítva az eltérés annyi, hogy itt nem lapokat, hanem rekordokat láncolunk. Egy vödör tehát egy rekordokból álló láncot jelent.

4.1.2. Nyitott címzés

A nyitott címzést használó módszerek általában – az itt bemutatásra kerülő technikák pedig különösen is – csak belső memóriás módszerként hasznosak. A rekordokat a $T[0 : M-1]$ táblában (tömbben) kívánjuk elhelyezni. A T tömböt tehát a h függvény értékkészletének elemeivel, a $[0, M-1]$ intervallum egészeivel indexeljük. A T elemei pedig rekordok. A K kulcsú rekordnak a $T[h(K)]$ helyet szánjuk, feltéve, hogy az nem foglalt.

Ellenkező esetben, amikor a $T[h(K)]$ cellában már egy korábban beillesztett rekord van, akkor valamilyen szisztematikus módon próbálunk helyet keresni a T további celláiban. A nyitott címzésű módszerek ütközésfeloldásra a $0, 1, \ldots, M-1$ számok egy $0, h_1(K), h_2(K), \ldots, h_{M-1}(K)$ *permutációját* használják. Pontosabban fogalmazva sorra végigpróbáljuk a $h(K) + h_i(K)$ sorszámú cellákat ($i = 0, 1, \ldots, M-1$) az első üres helyig, ahol a rekordot elhelyezzük. Itt az összeadás modulo M értendő, vagyis az összeg mindig 0 és $M-1$ közötti egésznek vehető. Permutációra azért van szükség, hogy a $h(K) + h_i(K)$ sorszámú cellák kiadják a tábla minden helyét. Ha nem találtunk üres helyet a sorozat mentén, akkor arra következtethetünk, hogy a tábla betelt, az új rekordot már nincs

hová tenni. A $h_i(K)$ számok jelölésében azért tüntettük fel K-t, hogy érzékeltessük: a próbasorozat függhet a kulcstól is.



Az előző bekezdésben tulajdonképpen elmondtuk a beillesztés algoritmusát. A keresés itt is egyszerű. Ha a keresett rekord kulcsa K, akkor sorra megnézzük a $h(K) + h_i(K)$ sorszámú cellákat ($i = 0, 1, \ldots$), amíg megtaláljuk a keresett rekordot, vagy üres cellához érünk, vagy pedig $i = M - 1$. Az utóbbi két esetben a keresett rekord nincs a táblában.

A törlés érdemi része is világos ezután. Megkeressük az eltávolítandó rekordot, majd töröljük a táblából. Ez utóbbi fázis a szokásosnál több körültekintést igényel. Később egy példa kapcsán még foglalkozunk ezzel a kérdéssel.

A módszerek összehasonlításához, hatékonyságuk elemzéséhez hasznosak lesznek a következő jelölések:

N – a táblában levő rekordok száma
M – a tábla celláinak száma
$\alpha = \frac{N}{M}$ – a telítettségi (betöltöttségi) tényező
C_N – a sikeres keresések (amikor a keresett rekord megtalálható T-ben) során megvizsgált cellák átlagos száma
C_N' – a sikertelen keresések (a keresett rekord nincs T-ben) során megvizsgált cellák átlagos száma.

A C_N meghatározásában az „átlagos" úgy értendő, hogy a táblában levő mindegyik elemre egyenlő eséllyel érkezik keresési igény. A C_N' esetében pedig arról van szó, hogy a $h(K)$ érték ugyanakkora eséllyel lehet a $0, 1, \ldots, M - 1$ számok bármelyike. Mindeme előkészületek után lássunk néhány fontosabb próbamódszert:

Lineáris próbálás
Itt $h_i(K) := -i$. A K kulcsú rekord beillesztése során a $h(K)$ című cellától indulva addig lépkedünk egyesével balra, amíg üres helyet nem találunk. A $T[0]$ cella után pedig, ha kell, a $T[M-1]$ cellával folytatjuk. A keresés költségére érvényesek a következők (nem bizonyítjuk):

$$C_N = \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

és

$$C'_N = \frac{1}{2}\left(1 + \left(\frac{1}{1-\alpha}\right)^2\right).$$

Néhány α értékre táblázatba foglaltuk a formulákból adódó keresési időket. Látható, hogy 80 százalékos telítettség felett a sikertelen keresések már elég sok időt igényelnek.

α	$2/3$	$0,8$	$0,9$
C_N	2	3	$5,5$
C'_N	5	13	$50,5$

Példa: Tegyük fel, hogy a kulcsok természetes számok, és a rekordjaink csak ebből az egyetlen mezőből állnak. Legyen a táblaméret $M = 7$, és legyen a hash-függvény $h(K) := K \pmod 7$. Tegyük fel még azt is, hogy az ütközések feloldására lineáris próbálást használunk. Így néz ki a tábla a 3, 11 és 9 kulcsok beillesztése után:

0	1	2	3	4	5	6
		9	3	11		

Ha most a $K = 4$ kulcsot szeretnénk beilleszteni, akkor mivel a $T[4]$, $T[3]$ és $T[2]$ helyek is foglaltak, hármat kell lépnünk a próbasorozat mentén. A négyes a $T[1]$-be kerül:

0	1	2	3	4	5	6
	4	9	3	11		

Töröljük most a 9 kulcsot. Első gondolatunk az lenne, hogy a sikeres keresés után felszabadítjuk a $T[2]$ helyet ugyanolyan NULL értéket adva neki, mint ami a még üres $T[0]$, $T[5]$ és $T[6]$ cellákban van. Ez azonban bajt okozhat, hiszen ezután a $K = 4$ kulcs keresése sikertelen lenne. A próbasorozat mentén NULL értékű cellához jutnánk, amiből azt hihetnénk, hogy a négyes nincs a táblában. A megoldás egy a NULL-tól különböző TÖRÖLT jel használata (ez lehet pl. $*$). A keresések során ezek a cellák átléphetők; úgy tekinthetjük őket, mintha foglaltak volnának. Új rekord beillesztésekor természetesen egy ilyen hely üresnek tekinthető, és ismét használható. A törlés után a helyzet tehát így alakul:

0	1	2	3	4	5	6
	4	*	3	11		

Ezután már nem lesz baj, amikor a négyest keressük. A $T[2]$-ben levő $*$ jelből tudni fogjuk, hogy tovább kell lépnünk a próbasorozat mentén. A törléssel kapcsolatban itt vázolt probléma általános a nyitott címzést használó módszereknél. A javasolt megoldás, a TÖRÖLT jel használata, működik más próbasorozatok esetén is.

A példa egyébként mutatja a lineáris próbálás fő hátrányát is. Ha sok rekord van a táblában, akkor hosszú, egybefüggő „csomók" alakulnak ki, megnövelve a keresési utak hosszát. Mondjuk a 18 keresésekor elég nagy utat kell megtenni, mire kiderül az igazság. Ezt a kellemetlen jelenséget *elsődleges csomósodásnak* (klasztereződésnek) szokás nevezni. A csomósodás oka abban van, hogy ha egy R' rekord beillesztése során elérjük a már tárolt R rekord keresési útját, akkor azt rendszerint végig is kell járnunk.

A lineáris próbálással ismerkedők gyakran kérdezik, hogy miért visszafelé lépkedünk a táblában, és nem előre. A magyarázat programozástechnikai természetű. Az eljárásokban (sokszor) ellenőriznünk kell, hogy a lépkedés során nem értük-e el a tábla végét. Ha visszafelé, a tábla eleje felé megyünk, akkor a nullával való egyenlőséget kell vizsgálni; ha a másik irányt választanánk, akkor egy M körüli egész szerepelne a tesztekben. Az előbbi típusú tesztek időigénye jóval kisebb, amitől a visszafelé lépkedő kód számottevően gyorsabb az előre menőnél.

Álvéletlen próbálás

Ezeknél a módszereknél a $0, h_1(K), h_2(K), \ldots, h_{M-1}(K)$ próbasorozat a $0, 1, \ldots, M-1$ számoknak egy a K kulcstól független álvéletlen permutációja. A sorozatnak természetesen gyorsan és hatékonyan reprodukálhatónak kell lennie, hiszen minden adatelérési műveletnél használni akarjuk. Ez a kikötés éppenséggel ellentétes a véletlenszerűséggel. Másfelől viszont ha a sorozat eléggé „véletlenszerű", elég szeszélyesen ugrabugrál, akkor a táblában nem tapasztalunk elsődleges csomósodást. A két törekvés egymással ellentétesnek tűnik, de mint példák mutatják, összebékíthetők.

Az álvéletlen próbálás sem mentes teljesen a csomósodástól, bár ez kisebb mértékben rontja a hatékonyságot, mint a lineáris próbálásra jellemző elsődleges csomósodás. Arról van szó, hogy ha a K és L kulcsokra $h(K) = h(L)$, akkor a K és L kulcsok teljes próbasorozata is megegyezik. Ha tehát sok azonos címre kerülő kulcs van, akkor a közös próbasorozatuk mentén alakul ki csomósodás. Ezt a jelenséget *másodlagos csomósodásnak* nevezzük.

Az álvéletlen próbálásra egy hasznos és mutatós példa a *kvadratikus maradék próba*. Legyen M egy $4k+3$ alakú prímszám, ahol k egy egész. Ekkor a próbaso-

rozat legyen

$$0, 1^2, -(1^2), 2^2, -(2^2), \ldots, \left(\frac{M-1}{2}\right)^2, -\left(\frac{M-1}{2}\right)^2.$$

Egy kis számolgatás (az $M = 19$-et javasoljuk az olvasónak) érzékelteti, hogy a sorozat tényleg elég szeszélyesen lépked. Elméleti érvek is szólnak a „véletlenszerűsége" mellett; ezeket most mellőzzük. Megmutatjuk viszont, hogy a sorozat a modulo M maradékosztályok egy permutációját adja. Ez könnyen következik az alábbi tényből.

Állítás: *Ha M egy $4k + 3$ alakú prímszám, akkor nincs olyan n egész, melyre* $n^2 \equiv -1 \pmod{M}$.

Bizonyítás: Indirekt érvelve tegyük fel, hogy n egy egész szám és $n^2 \equiv -1 \pmod{M}$. Ekkor

$$-1 \equiv (-1)^{\frac{M-1}{2}} \equiv n^{2\frac{M-1}{2}} \equiv n^{M-1} \equiv 1 \pmod{M}.$$

Az utolsó lépésnél a kis Fermat-tételt használtuk. Az elejét a végével összevetve $-1 \equiv 1 \pmod{M}$ adódik, ami képtelenség. □

Nyilvánvaló mármost, hogy ha $0 \leq i < j \leq \frac{M-1}{2}$, akkor $i^2 \not\equiv j^2 \pmod{M}$. Ugyanis a $j^2 - i^2 = (j-i)(j+i)$ felbontás egyik tényezője sem lehet osztható M-mel, tehát a szorzatuk sem. Ugyanígy kapjuk, hogy $-i^2 \not\equiv -j^2 \pmod{M}$. Az $i^2 \equiv -j^2 \pmod{M}$ kongruenciából pedig $(ij^{-1})^2 \equiv -1 \pmod{M}$ következne, ahol j^{-1} jelöli a j multiplikatív inverzét modulo M. Ez a kongruencia pedig az előző állítás szerint nem lehetséges. A próbasorozatban tényleg nincs ismétlődés.

Ami a keresés költségét illeti, a legjobb változatok időigénye így alakul (nem bizonyítjuk):

$$C_N \approx 1 - \log(1-\alpha) - \frac{\alpha}{2}$$

a sikeres keresésre, és

$$C'_N \approx \frac{1}{(1-\alpha)} - \alpha - \log(1-\alpha)$$

a sikertelen keresés során megvizsgált cellák átlagos számára. Ezek az összefüggések valamivel általánosabban érvényesek az olyan módszerekre, amelyekre $h_i(K) = f_i(h(K))$; vagyis ahol a $h(K)$ érték már az egész próbasorozatot meghatározza.

Kettős hash-elés (G. de Balbine, J. R. Bell, C. H. Kaman, 1970 körül.)
A recept lényege, hogy a h mellett egy másik h' hash-függvényt is használunk. Utóbbival szemben követelmény, hogy a $h'(K)$ értékek relatív prímek legyenek az M táblamérethez. A K kulcs próbasorozata ekkor $h_i(K) := -ih'(K)$. Ha M és $h'(K)$ relatív prímek, akkor a $0, -h'(K), -2h'(K), \ldots, -(M-1)h'(K)$ sorozat elemei mind különbözők modulo M, így alkalmas lesz próbasorozatnak. E módszerek fontos sajátossága, hogy a különböző K és K' kulcsok próbasorozatai jó eséllyel akkor is különbözők lesznek, ha $h(K) = h(K')$. A legjobb ismert implementációk időigénye (empirikus adatok alapján)

$$C_N \approx \frac{1}{\alpha} \log \frac{1}{(1-\alpha)} \quad \text{és} \quad C'_N \approx \frac{1}{1-\alpha}.$$

A kettős hash-elés hatékony, a gyakorlatban is jó viselkedést mutató módszer. Ezzel kapcsolatos további tapasztalati észrevételek a következők:

1. A kettős hash-elés kiküszöböli mindkétféle csomósodást.
2. Sikertelen keresés esetén minden érdekes α-ra gyorsabb, mint a lineáris próbálás.
3. Sikeres kereséskor csak az $\alpha \geq 0,8$ tartományban lesz gyorsabb a lineáris próbálásnál.

Végezetül bizonyítás nélkül megemlítünk néhány tényt a vödrös hash-elés belső memóriás változatáról. Ekkor rekordokat (cellákat) láncolunk. Egy vödör tehát cellákból álló lánc lesz. Jelölje M a vödörkatalógus méretét. Tegyük fel, hogy a rekordok a kulcs szerint nem csökkenő sorrendben vannak a listájukra fűzve. Ekkor igazolható, hogy ha $0 < \alpha < 1$, akkor

$$C_N \approx 1 + \frac{1}{2}\alpha \quad \text{és} \quad C'_N \approx 1 + \frac{1}{2}\alpha^2.$$

A láncolás általában jóval helyigényesebb a nyitott címzésű módszereknél. A megvizsgált cellák számát tekintve – ezt mérik a C_N és C'_N mennyiségek – viszont hatékonyabb azoknál mind a sikeres, mind a sikertelen kereséseket illetően. Valódi időben mérve az összevetés már nem ennyire egyértelmű, mert a listák kezelése nehézkesebb, mint a tömbszerű tábláké.

Feladat: Igazoljuk a belső memóriás láncolás sikeres keresésének idejét megadó formulát.

4.2. Hash-függvények

A hash-elést használó módszerek hatékonysága nagymértékben függ a h hash-függvény minőségétől. A h függvénnyel szemben két alapvető követelmény van:

legyen könnyen (gyorsan) számítható, és minél kevesebb ütközést okozzon.

Az első követelmény magától értetődő: a módszer alkalmazása során minden rekordelérés egy $h(K)$ érték kiszámítása alapján történik, ahol K egy kulcs.

A második követelmény elég nehezen megfogható, mert a gyakorlatban előforduló kulcshalmazok egyáltalán nem véletlenszerűek (a fejezet elejének példájához kapcsolódva mondjuk a Kovács név sokkal gyakoribb, mint az Eördögh). Kevés elméleti eredmény van, viszont jelentős tapasztalati hátterű ismeret halmozódott fel. Ilyen hasznos empirikus tanácsok például, hogy $h(K)$ értéke lehetőleg a K kulcs minden bitjétől függjön, és a h értékkészlete a teljes $[0, M-1]$ címtartomány legyen. Fontosságát tekintve két módszer emelkedik ki a mezőnyből. Ezeket ismertetjük a következőkben.

Osztómódszer

Legyen $h(K) := K \pmod M$, ahol M a tábla vagy a vödörkatalógus mérete. Itt és a továbbiakban is feltesszük, hogy a kulcsok egész számok. Ez nem jelent lényegi korlátozást. A gyakorlatban előforduló kulcsok általában könnyen és gyorsan konvertálhatók egészekké. Legtöbbször arról van szó, hogy valamilyen bitsorozatot számként értelmezünk. A $h(K)$ számítása gyors és egyszerű: mindössze egy maradékos osztást igényel. A lineáris próbálásnál szereplő példában az osztómódszert alkalmaztuk.

Az osztómódszer használatakor M, a tábla mérete sem teljesen közömbös. Például ha M a 2 egy hatványa, akkor $h(K)$ csak a kulcs utolsó néhány bitjétől függ. A jó M értékeket illetően van egy széles körben elfogadott recept, amit a szakma *D. E. Knuth javaslata*ként tart számon. Eszerint megfelelő, ha M-et prímnek választjuk, úgy, hogy M nem osztja $r^k \pm a$-t, ahol r a karakterkészlet elemszáma (pl. 128, vagy 256) és a, k „kicsi" egészek.

Az, hogy M prím, lényeges feltétel a kvadratikus maradék próbánál. Ugyancsak előnye az ilyen számoknak, hogy könnyű hozzájuk relatív prím számot találni. Az utóbbi tulajdonság a kettős hash-elésnél jut szerephez.

Szorzómódszer

Ennél az eljárásnál egy rögzített β paraméter használatos, ami egy valós szám. Ennek birtokában legyen

$$h(K) := \lfloor M \cdot \{\beta K\} \rfloor.$$

A formulában $\{x\}$ jelöli az x valós szám törtrészét. Szemléletesen az történik, hogy $\{\beta K\}$ kiszámításával a K kulcsot „véletlenszerűen" belőjük a $[0, 1)$ intervallumba, majd az eredményt felskálázzuk a címtartományba.

Hatékonyan számítható speciális eset a következő: legyen a táblaméret $M = 2^t$, jelölje w a gépünk szókapacitását (pl. $w = 2^{32}$), és legyen A egy a w-hez relatív prím egész. Ekkor $\beta = \frac{A}{w}$ választás mellett $h(K)$ igen jól számolható. A számok bináris ábrázolásával dolgozva lényegében egy szorzást és egy eltolást kell elvégezni.

A szorzómódszer (és az osztómódszer is) jól viselkedik számtani sorozatokon, azaz $\{K, K + d, K + 2d, \ldots\}$ alakú kulcshalmazokon. Ilyenek könnyen adódnak gyakorlati adatkezelési feladatokból, amint azt a *termék1, termék2, termék3,* ... és a hasonló szerkezetű kulcshalmazok mutatják. Megmutatható, hogy a $h(K), h(K+d), h(K+2d) \ldots$ sorozat közelítőleg számtani sorozat lesz, azaz h jól „szétdobja" a kulcsok számtani sorozatait. Ennek a ténynek szép elméleti háttere van. Tegyük fel, hogy β egy irracionális szám, és nézzük a $0, \{\beta\}, \{2\beta\}, \ldots, \{n\beta\}$ sorozat eloszlását $[0, 1)$-ben. Nagy n értékre ez egyenletes: egy z hosszúságú $(0 < z < 1)$ részintervallumba körülbelül zn elem esik, ha $n \to \infty$. Ez adódik az alábbi csodaszép eredményből.

Tétel (T. Sós Vera, 1957): *Legyen β irracionális szám, és nézzük a 0, $\{\beta\}$, $\{2\beta\}$,..., $\{n\beta\}$ pontok által meghatározott $n + 1$ részintervallumot $[0, 1)$-ben. Ezek hosszai legfeljebb 3 különböző értéket vehetnek fel, és $\{(n + 1)\beta\}$ a leghosszabbak egyikét fogja két részre vágni.* □

Ismert még, hogy a $[0, 1)$-beli számok közül a legegyenletesebb eloszlást a $\beta = \phi^{-1} = \frac{\sqrt{5}-1}{2} = 0.618033988\ldots$ és a $\beta = \phi^{-2} = 1 - \phi^{-1}$ értékek adják. Ekkor ha $\{(n + 1)\beta\}$ a korábbi osztópontok által meghatározott I részintervallumba esik, és azt a, valamint b hosszúságú részekre osztja, akkor $\frac{1}{2} < \frac{a}{b} < 2$ mindig teljesül. Eszerint érdemes a szorzómódszernél az A egészet úgy választani, hogy $\frac{A}{w}$ közel legyen ϕ^{-1}-hez. A β ilyen választásával kapott módszereket *Fibonacci-hash-elésnek* nevezzük.

A kettős hash-elés második függvénye

A kettős hash-elésnél olyan h' függvényre van szükség, melynek értékei a $[0, M - 1]$ intervallumba esnek, és relatív prímek az M táblamérethez. Ha M prím, akkor a következő függvény megteszi:

$$h'(K) := K \pmod{M - 1} + 1.$$

Az első tag 0 és $M - 2$ közé eső egész, ezért a h' értékei 1 és $M - 1$ között lesznek, sőt az értékkészlete éppen az $[1, M - 1]$ intervallum egészeiből áll. Ebből két hasznos dolog következik. Egyrészt az, hogy $h'(K)$ és M relatív prímek. Másrészt az is teljesül, hogy elég sok – szám szerint $M - 1$ – különböző próbasorozatot ad.

4.3. Hash-elés kontra keresőfák

Két erőteljes és gazdag módszercsaládot ismertünk meg a keresés, beszúrás, törlés és az ezekre épülő műveletek megvalósítására; divatos kifejezéssel élve: dinamikus halmazok kezelésére. Az egyik megközelítés a hash-elt szervezés, a másik pedig a keresőfa adatszerkezet. Természetesen vetődik fel a kérdés, hogy milyen szempontok szerint válasszunk a kettő közül. Ez felmerülhet úgy, hogy nekünk kell egy dinamikus halmazokat kezelő programot írni, de úgy is, hogy egy készen kapható rendszer (pl. adatbáziskezelő) által nyújtott lehetőségek közül kell választani. Három szempontot említünk, amelyek segítenek az eligazodásban.

Átlagos viselkedés – legrosszabb eset

A hash-elt szervezésnél az alapműveletek (keresés, beszúrás, törlés) átlagos értelemben gyorsabbak, mint a keresőfáknál. Ezt tükrözi az a tény, hogy a hash-elésnél rögzített $\alpha < 1$ telítettségi tényezőnél a megvizsgált lapok, illetve cellák száma átlagosan konstans korlát alatt marad. Számos alkalmazásnál éppen az átlagos sebesség, ami igazán számít. Például az ügyviteli alkalmazások egy jelentős részénél csak az a fontos, hogy nagy mennyiségű teendőt összességében kedvező idő alatt elvégezzünk, és nem érdekes, ha ezek közül néhány hosszabb ideig tart, mint a többi. Az ilyen esetekben a hash-elés felé billen a mérleg nyelve.

Vannak ezzel szemben olyan helyzetek, ahol a legrosszabb, legkedvezőtlenebb esetekben is elég gyorsnak kell lenni. A legdrámaibb példák ebből a szempontból az ún. *valós idejű* feladatok. Ezek jellemzője, hogy a programnak meg kell felelnie valamilyen külső folyamatok feszített időbeli kényszereinek. Mondjuk egy erőmű kazánjának szelepeit vezérlő rendszertől elvárjuk, hogy a megnövekedett hőmérsékletre még azelőtt reagáljon, nyissa ki a szelepeket, mielőtt szétrobban a kazán. Groteszk lenne a katasztrófa után azzal takarózni, hogy a program éppen egy hosszú vödörben kotorászott valami adat után. Az ilyen körülmények között adódó dinamikus halmazokat keresőfákban kell tárolnunk, mert azok a legrosszabb esetek idejére is garanciát nyújtanak.

Rendezés a felhasználói igényekben

Előfordul, hogy az adatok rendezésére vonatkozó feltételek komoly szerepet játszanak a felhasználói igényekben, például sok MIN, MAX, és/vagy TÓLIG-típusú kérdésre kell válaszolni. Ekkor a keresőfák ajánlhatók, hiszen jó megoldásokat nyújtanak az ilyen feladatokra. A hash-elt szervezések nem, vagy csak meglehetősen bonyolult kiegészítő megoldásokkal tudják figyelembe venni a rendezettséget.

Dinamika

A hash-elt szervezés nem támogatja az állomány nagyon dinamikus növekedését. Ha a tábla betelik, vagy ha α már túl nagy, akkor új, nagyobb táblával (vödörkatalógussal) újra kell szervezni az állományt. A keresőfák ezzel szemben rendkívül rugalmasan, tág határok között tudják változtatni a méretüket. Ha tehát úgy látjuk, hogy a tárolandó állomány mérete erőteljesen változni fog, vagy nem tudunk a méretre realisztikus felső korlátot adni, akkor inkább a keresőfák ajánlhatók.

4.4. Szekvenciális keresés

Gyakran előfordul, hogy egy állomány (tömb, lista, stb.) szegényes szerkezetű. Ez alatt azt értjük, hogy nincs benne elég rendszer, és ezért a keresésre nincs jobb módszerünk, mint a szerkezetet „elejétől a végéig" bejárni, vagy legalábbis addig, amíg a keresett adatot meg nem találjuk. Az ilyen esetekben *szekvenciális keresésről* beszélünk. A modell, amit használunk, a *rekordokból álló tábla*.

R_1	R_2	R_3	$\ldots$	R_{N-1}	R_N

A feladat az R rekord megtalálása a táblában, ha egyáltalán benne van. A keresés úgy történik, hogy a tábla elejétől kezdve sorra összehasonlítjuk az R rekordot az R_i rekordokkal ($i = 1, 2, \ldots$). A keresés befejeződik, ha $R = R_i$, vagy $i = N$. A keresés költsége a tábla megvizsgált celláinak száma.

Ha semmiféle további feltevéssel nem élünk, akkor a sikeres keresés átlagos költsége

$$\frac{1 + 2 + \cdots + N}{N} = \frac{N+1}{2}.$$

Az átlag úgy értendő, hogy az $R = R_i$ események egyenlő esélyűek. Sikertelen kereséskor, amikor az R rekord nincs a táblában, a költség mindig N.

Ha a tábla a rekordokat egyértelműen meghatározó kulcs szerint rendezett, akkor sikeres keresésnél a helyzet változatlan, $\frac{N+1}{2}$ az átlagos költség. A sikertelen keresés időigénye csökkenhet, hiszen a táblát csak akkor kell végignézni, ha R kulcsa nagyobb az R_N kulcsánál. Beszélhetünk átlagos költségről is: a beszúrásos rendezés elemzésében használt feltételhez hasonlóan tegyük fel, hogy az R egyenlő eséllyel kerül az R_i rekordok által meghatározott $N + 1$ rendezési intervallum bármelyikébe. Ekkor a megvizsgált cellák átlagos száma

$$\frac{1 + 2 + \cdots + N - 1 + N + N}{N+1} = \frac{N}{2} + \frac{N}{N+1} < \frac{N}{2} + 1.$$

Tegyük fel a továbbiakban, hogy csak sikeres keresésekkel van dolgunk, és legyen p_i annak a valószínűsége, hogy az R_i rekordot keressük. Ekkor $p_1 + p_2 + \cdots + p_N = 1$, és a keresés várható költsége

$$C_N = p_1 + 2p_2 + 3p_3 + \cdots + Np_N.$$

Feladat: Mutassuk meg, hogy adott valószínűségek mellett C_N akkor minimális, ha $p_1 \geq p_2 \geq \ldots \geq p_N$. (Ha $i < j$ és $p_i < p_j$, akkor az $R_i \leftrightarrow R_j$ csere után C_N kisebb lesz.)

A feladat szerint a rekordokat a keresési gyakoriságok szerint nem növekvő sorrendben érdemes a táblában tartani. Előfordul, hogy a $\{p_i\}$ eloszlást, vagy annak elég jó közelítését ismerjük, és ezért ki tudjuk alakítani az optimális sorrendet. Olyan alkalmazásoknál fordul ez elő, ahol a tábla statikus, a tartalma nem változik, és kizárólag keresési igények merülnek fel. Ilyen alkalmazásra elég jó példa egy archív szövegeket (mondjuk régi folyóiratok tartalmát) tároló rendszer. Nézzük meg ezután, hogy miként alakul a C_N értéke néhány érdekes eloszlás esetén, amikor a rekordok a p_i értékek szerint nem növekvő sorrendben vannak a táblában.

Egyenletes eloszlás: Itt $p_i = \frac{1}{N}$ minden i-re. Ekkor, ahogy azt az előbb már láttuk, $C_N = \frac{N+1}{2}$. Átlagosan tehát a tábla celláinak a felét kell megnézni.

Egy nagyon ferde eloszlás: Legyen $p_i = \frac{1}{2^i}$ ha $1 \leq i \leq N-1$, és $p_N = \frac{1}{2^{N-1}}$. A kupacépítés elemzésénél használt háromszög alakú táblázat összegezésével belátható, hogy C_N értéke nagyon kicsi: $C_N = 2 - \frac{1}{2^{N-1}}$. Ez nem meglepő, hiszen a p_i sorozat rohamosan csökken.

Zipf eloszlás: Jelölje H_n az n-edik harmonikus számot:

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}.$$

Ahogy azt a gyorsrendezés elemzésénél már említettük, $H_n = \log n + \gamma + o(1)$, ahol $\gamma = 0.55721\ldots$ az Euler-állandó. A Zipf eloszlás valószínűségei $p_i = \frac{c}{i}$, ahol $c = \frac{1}{H_N}$. A Zipf eloszlás előfordul több, az adatkezelés szempontjából érdekes összefüggésben. Közelítőleg ilyen eloszlást követ például egy átlagos (angol nyelvű) szövegben az i-edik leggyakoribb szó gyakorisága. Egyszerűen adódik, hogy

$$C_N = \sum_{i=1}^{N} ip_i = \sum_{i=1}^{N} i\frac{c}{i} = Nc = \frac{N}{H_N}.$$

A keresés várható költsége tehát $N/\log N$ körül van.

80-20 szabály: Arról a számítógépes gyakorlatban megfigyelt, sok esetben érvényes jelenségről van szó, hogy az adatelérési igények 80%-a a rekordoknak körülbelül csak 20%-át érinti. Erre a 20%-ra ugyanez a szabály érvényes. Ebből következik, hogy az igények 64%-a az állománynak mindössze 4%-át érinti.

A 80-20 szabály szerint viselkedő eloszlásra minden $1 \leq m \leq N$ esetén

$$\frac{p_1 + p_2 + \cdots + p_{0,2m}}{p_1 + \ldots + p_m} \approx 0,8.$$

E követelményeknek megközelítően eleget tevő eloszlást kapunk a $p_i = \frac{c}{i^{1-\vartheta}}$ választással, ahol

$$\vartheta = \frac{\log 0,8}{\log 0,2} \approx \frac{1}{7}, \quad c = \frac{1}{H_N^{(1-\vartheta)}} \quad \text{és} \quad H_N^{(1-\vartheta)} = 1 + \frac{1}{2^{(1-\vartheta)}} + \cdots + \frac{1}{N^{(1-\vartheta)}}.$$

A sikeres keresés várható idejére megmutatható, hogy $C_N \approx 0,122N$. Egy ilyen eloszlást követő állományban az optimális sorrend mellett a keresés mintegy négyszer gyorsabb, mint a rekordok (egyenletes eloszlás szerinti) véletlen sorrendje esetén.

Önszervező módszerek

Mit tehetünk abban az esetben, ha a p_i keresési valószínűségeket nem ismerjük, vagy esetleg azok idővel változnak? Ilyenkor a sikeres keresés után érdemes változtatni a rekordok sorrendjén. Az S-fáknál már láttunk példát arra, hogy visszacsatoláson alapuló egyszerű változtatások jelentősen javíthatják egy adatszerkezet hatékonyságát. A jelen esetben a keresések sorozatát úgy is felfoghatjuk, hogy mintát veszünk a $\{p_i\}$ eloszlásból, vagyis nézzük az $R = R_i$ események q_i tapasztalati gyakoriságát. A q_i gyakoriság nő, ha az éppen megtalált rekord R_i, ezért ekkor R_i-t előbbre visszük a táblában, feltéve, hogy $i > 1$. Két alapvető ötletet említünk.

(a) A keresett (és megtalált) R_i elemet a tábla elejére visszük. Az eredmény:

R_i	R_1	R_2	...		R_N

(b) A keresett (és megtalált) R_i elemet felcseréljük a megelőzővel.

R_1	...	R_i	R_{i-1}	...		R_N

Ha az eloszlás időben változik, akkor az (a) megoldás ajánlatos. Ha a $\{p_i\}$ eloszlás stabil, azaz időben nem változik, akkor a (b) heurisztika eredményesebb. A (b) változatról azt sejtik, hogy optimális megoldást ad az olyan esetekben, amikor egy ismeretlen statikus eloszlással van dolgunk, és nem áll módunkban nyilvántartani a q_i elérési gyakoriságokat.

5.

Információtömörítés

Arra törekszem, hogy tíz mondatban mondjam el mindazt, amire másoknak egy egész könyv kell. FRIEDRICH NIETZSCHE

Manapság óriási mennyiségű információt tárolunk a gépeinkben, és küldünk a szélrózsa minden irányába a hálózatok mentén. A tárolás és különösen a továbbítás költsége erősen függ a szóban forgó adatok mennyiségétől. Emiatt természetes az igény, hogy bizonyos esetekben adatainkat minél tömörebb, kompaktabb formában ábrázoljuk. Az *adattömörítés* mára valóságos tudományággá vált. Se szeri, se száma a különböző információfajták (pl. szöveg, kép, hang) összenyomására, illetve kibontására szolgáló módszereknek. Itt a legelső, legegyszerűbb eljárást, a ma már klasszikusnak számító Huffman-kódolást ismertetjük, majd a széles körben használatos Lempel–Ziv–Welch-módszer ötletét mutatjuk be.

5.1. A Huffman-kód

Tegyük fel, hogy egy a $b_1, \ldots, b_n$ betűkből álló szöveget szeretnénk bitsorozatként kódolni. A b_i betűk valamely abc jelei, és egy ezekből szerkesztett hosszú (többnyire n-nél sokkal hosszabb) sorozat gazdaságos kódolása a célunk. Az egyes betűk kódjait (a nekik megfelelő bitsorozatokat) úgy akarjuk megválasztani, hogy a szöveg kódjának hossza minimális legyen. Ismert még az egyes betűk $q_1, \ldots, q_n$ gyakorisága a szövegben. Ha *uniform*, tehát ugyanolyan hosszú sorozatokat használunk a kódolásra, akkor nem sok játékterünk marad, hiszen a gyakoriságoktól függetlenül (feltéve, hogy ezek mind pozitívak) $\lceil \log_2 n \rceil$ a legrövidebb lehetséges kódhosszúság egy betűre. Ha megengedünk eltérő hosszú kódszavakat, akkor a dekódolással, a szövegnek a bitsorozatból való visszanyerésével lehet probléma. Olyan kódot szeretnénk tehát, ahol a betűk kódszavainak hossza nem feltétlenül

azonos, és a kód egyértelműen visszafejthető. Egy lehetséges megoldást kínál a következő definíció:

Definíció: *Egy bitsorozatokból álló kód* prefix kód, *ha egy betű kódja sem prefixe (kezdőszelete) egy másik kódjának. Formálisan: ha x és y két különböző betű kódja, akkor nincs olyan z bitsorozat, melyre $xz = y$.*

Egy prefix-tulajdonságú kóddal leírt üzenet egyértelműen visszafejthető. Az üzenet elejétől addig megyünk a bitek mentén, amíg egy betű kódszavát kapjuk. A prefix-feltétel miatt csak ez lehetett az első betű. Innen ugyanígy folytatva sorban megfejthetjük a további betűket.

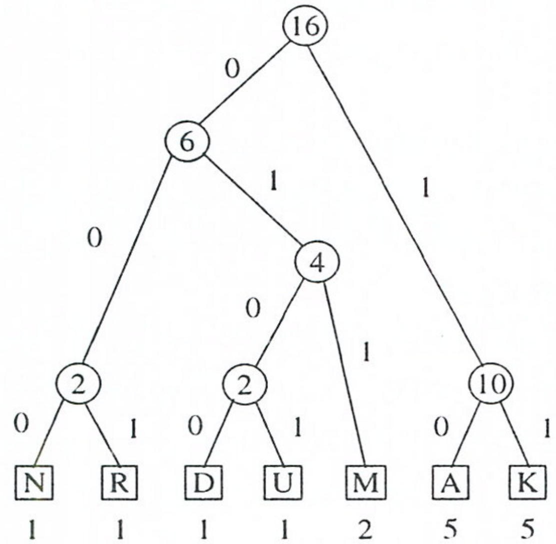
Egy prefix kódhoz természetes módon bináris fát rendelhetünk. Vegyünk először egy elég nagy teljes bináris fát. Ennek balra menő éleit címkézzük nullával, a jobbra menőket pedig egyessel. Ebben a fában egy a gyökérből lefelé menő út kölcsönösen egyértelműen leírható egy bitsorozattal, az érintett élek címkéinek sorozatával. Nézzük meg a b_i betűk kódjainak megfelelő utakat. Ezek alsó végpontjait címkézzük meg b_i-vel. A prefix-tulajdonság azt jelenti, hogy b_i nem őse b_j-nek, ha $i \neq j$. Ezért ha az előző utak által nem érintett csúcsokat és az ezekhez illeszkedő éleket eldobjuk a fából, akkor egy olyan bináris fát kapunk, melynek levelei pontosan a $b_1, \ldots, b_n$ címkéjű csúcsok. Megfordítva: egy bináris fából, melynek levelein a b_i címkék vannak (különböző leveleken különbözők), a kézenfekvő módon egy prefix kódot kapunk.

Az előző megfeleltetést figyelembe véve a szövegtömörítési probléma prefix kód alkalmazása esetén így fogalmazható: azon bináris fák között, melynek levelei $b_1, \ldots, b_n$, keressünk olyat, amelyre a $\sum q_i h(b_i)$ összeg minimális, ahol egy x csúcsra $h(x)$ a gyökértől x-ig vezető úton bejárt élek száma. Valóban, $h(b_i)$ éppen a b_i betű bitkódjának a hossza, így az összeg éppen a kódolt szöveg hosszát méri.

Az előző értelemben optimális konstrukció a *Huffman-fa*. Az algoritmus egy bináris fát épít, melynek a csúcsait pozitív valós számokkal címkézzük. Kezdetben n izolált csúcspontunk van. Ezek felelnek meg a b_i betűknek, és ezek lesznek végül a Huffman-fa levelei. A b_i-nek megfelelő csúcs címkéje q_i. Az általános lépés a következő. Tegyük fel, hogy már megépítettük az $S_1, \ldots, S_k$ fákat, ezek gyökérpontjai $x_1, \ldots, x_k$, utóbbiak címkéi az $r_1, \ldots, r_k$ számok. Ekkor vesszük a két minimális címkéjű gyökeret (legyenek ezek x_i és x_j). Ezek fölé egy új y gyökérpontot teszünk, melynek fiai x_i és x_j. Az y címkéje $r_i + r_j$. Ezzel a fák száma eggyel csökken. Megállunk, ha már csak egy fa marad. Összesen tehát $n - 1$ ilyen összevonó lépés szükséges.

Példaként tekintsük a KAKUKKMADARAMNAK szó kódolását. Ebben hét betű szerepel, a legrövidebb lehetséges uniform kódhossz betűnként 3 bit. A 16 betűs szó kódolása így 48 bitet igényel. Nézzük, mit ad a Huffman-kód. A betűk

gyakoriságai (a q_i értékek): A: 5, K:5, M: 2, U: 1, D: 1, R: 1, N: 1. Egy lehetséges kezdés, hogy először az N, és R csúcsokat (fákat) vonjuk össze, majd a D és U csúcsokat, utána az utóbbi fa gyökerét az M csúccsal. Az ábra egy ilyen fát mutat.



A betűk kódjai: K: 11, A: 10, M: 011, U: 0101, D: 0100, N: 000, R: 001. A szó kódjának a hossza így 40 bit, ami megtakarítást mutat az uniform kódoláshoz képest.

A módszer egyszerűsége mellett figyelemre méltó, hogy a prefix kódok közül a lehető legjobb tömörítést eredményezi. Nevezetesen igaz a következő:

Tétel: *A Huffman-fa optimális. Pontosabban fogalmazva, a Huffman-fa esetén az* $I = \sum q_i h(b_i)$ *összeg minimális azon bináris fák között, amelyek levelei* $b_1, \ldots, b_n$.

Bizonyítás: A Huffman-fa által adott I érték legyen $H(q_1, q_2, \ldots, q_n)$. Tegyük fel még, hogy $q_1 \leq q_2$ a két legkisebb gyakoriság. A fa konstrukciójából adódik, hogy

$$H(q_1, \ldots, q_n) = H(q_1 + q_2, q_3, \ldots, q_n) + q_1 + q_2.$$

Indoklásul megjegyezzük, hogy az első összevonó lépés után $n-1$ gyökérpont van, melyek címkéi rendre $q_1 + q_2, q_3, \ldots, q_n$. Az ezek feletti fa éppen egy Huffman-fa, melynek I-értéke $H(q_1+q_2, q_3, \ldots, q_n)$, amiből az eredeti fa I-értéke $q_1 + q_2$ hozzáadásával kapható. Jelölje $Opt(q_1, q_2, \ldots, q_n)$ a q_i gyakoriságok esetén elérhető optimális I-értéket.

Lemma: *Legyen továbbra is* $q_1 \leq q_2$ *a két legkisebb gyakoriság. Ekkor*

$$Opt(q_1, q_2, \ldots, q_n) = Opt(q_1 + q_2, q_3, \ldots, q_n) + q_1 + q_2.$$

A lemma bizonyítása: Tegyünk először néhány megállapítást az optimális fákról. Egy ilyenben feltehető, hogy minden belső csúcsnak két fia van. Ha ugyanis az y csúcsnak csak egy fia volna, akkor y-t törölhetnénk az egy szem fiát téve a helyére, ezzel csökkentve I értékét. Legyen most x egy levél az optimális fánkban, melyre $h(x)$ a lehető legnagyobb. Az előbbi fejtegetés szerint x-nek van a fában egy y testvére. Feltehető ezután, hogy x és y címkéi q_1 és q_2, hiszen a megfelelő cserék nem növelik I értékét. Töröljük egy pillanatra az x és y csúcsokat, és írjuk a keletkező új levélbe a $q_1 + q_2$ címkét. A fa I-értéke legyen J. Kapjuk azonnal, hogy $Opt(q_1, q_2, \ldots, q_n) = J + q_1 + q_2$, amiből az optimalitás miatt következik, hogy az új fának is optimálisnak kell lennie a $q_1 + q_2, q_3, \ldots, q_n$ gyakoriságokra vonatkozóan. Ez azért igaz, mert ha J-nél kisebb értékű megoldás is lenne, akkor ezzel az eredeti fán is tudnánk javítani. Mindebből arra jutunk, hogy

$$Opt(q_1, q_2, \ldots, q_n) = Opt(q_1 + q_2, q_3, \ldots, q_n) + q_1 + q_2,$$

ami éppen a lemma állítása. □

Ezután a tétel könnyű indukció n szerint. Az $n = 2$ esetben a Huffman-fa nyilván optimális, vagyis tetszőleges p és q gyakoriságokkal teljesül, hogy $Opt(p, q) = H(p, q)$. Tegyük fel mármost, hogy legfeljebb $n - 1$ betű esetén igaz a tétel állítása, és legyen $b_1, \ldots, b_n$ egy n elemű betűkészlet, melyre a gyakoriságok $q_1 \leq q_2 \leq \ldots \leq q_n$. Az indukciós feltevés szerint $Opt(q_1 + q_2, q_3, \ldots, q_n) = H(q_1 + q_2, q_3, \ldots, q_n)$. A H-ra vonatkozó kiemelt formula és a lemmabeli formula jobboldalán levő mennyiségek tehát egyenlők. Ebből következik, hogy a baloldalak is megegyeznek:

$$Opt(q_1, q_2, q_3, \ldots, q_n) = H(q_1, q_2, q_3, \ldots, q_n).$$

□

A módszer hátránya, hogy alkalmazásához kétszer kell végigolvasni a kódolandó szöveget. Az első olvasatban meghatározzuk a $q_1, q_2, \ldots, q_n$ gyakoriságokat, ezek alapján felépítjük a Huffman-fát, és a második menetben sorra elkódoljuk a szöveg betűit. Úgy is fogalmazhatunk, hogy a kódolás csak akkor kezdődhet, ha már az egész szöveg a rendelkezésünkre áll. Ez egy sor alkalmazásnál eléggé kényelmetlen. Vannak a Huffman-módszernek *dinamikus* változatai. Ezek a szöveg eddig olvasott részéből számolnak gyakoriságokat, és az így kapott fa szerint kódolják a következő betűt. Ennél a megközelítésnél a kódszavak fája betűről betűre változhat. Az ilyen egymenetes módszerek nem érnek el ugyan optimális összenyomást, de a tömörítő képességük még mindig elég kedvező. Ugyanakkor gyorsak, és megfelelnek az interaktív alkalmazások igényeinek. Ilyen megoldás van néhány régebbi UNIX-változat (mint pl. a 4.2 BSD UNIX) *compact* eljárásában.

5.2. A Lempel–Ziv–Welch-módszer

A Huffman-algoritmusoktól gyökeresen eltérő szemléletű módszercsaládot javasolt A. Lempel és J. Ziv a hetvenes évek második felében. Ötleteiknek egy szép és egyszerű megvalósítását dolgozta ki T. Welch 1984-ben. A Lempel–Ziv–Welch-módszernek nevezett eljárás ma az egyik legnépszerűbb információtömörítő megoldás. A módszer a Huffman-algoritmussal szemben nem betűnként kódolja az üzenetet. A szöveg bizonyos szavaiból *szótárat* épít. Ez szavak egy halmaza, amit S-sel fogunk jelölni. A szótárról három dolgot tételezünk fel:

– az egybetűs szavak, azaz Σ elemei mind benne vannak S-ben;
– ha egy szó benne van a szótárban, akkor annak minden kezdődarabja is benne van;
– a szótárban tárolt szavaknak fix hosszúságú kódjuk van; az $x \in S$ szó kódját $c(x)$ jelöli.

A gyakorlatban a kódok hosszát 12-15 bitnek érdemes választani. Az algoritmus az összenyomni kívánt szöveget S-beli szavak egymásutánjára bontja. A kódolás eredménye, az összenyomott szöveg az így kapott szavak kódjainak a sorozata. Az eredeti szöveg olvasásakor gyakorlatilag egyidőben épül/bővül az S szótár és alakul ki a felbontás. A helymegtakarítás, a tömörítés abból adódik, hogy sokszor helyettesítünk hosszú szavakat a rövid kódjaikkal. Az S építését és a szöveg felbontását is szinte banálisan egyszerű mohó stratégiával kezeli a módszer. Nyoma sincs benne teljes körű optimalizálásnak, ami a Huffman-algoritmus jellemzője. Pusztán az éppen vizsgált kis szövegdarab ismeretében alakulnak ki az érdemi döntések. A megoldás mégis rendkívül hatékony akár a sebességet tekintjük, akár az összenyomás mértékét.

A szótár egyik szokásos tárolási módja a szófa adatszerkezet. Az $x \in S$ szó $c(x)$ kódja úgy is tekinthető, mint egy hivatkozás (mutató) az x-nek az S-beli előfordulására. A szöveg összenyomása úgy történik, hogy amikor az olvasás során egy $x \in S$ szót találunk, aminek a következő Y betűvel való folytatása már nincs S-ben, akkor $c(x)$-et kiírjuk a kódolt szövegbe. Az xY szót felvesszük az S szótárba. A szó $c(xY)$ kódja a legkisebb még eddig az S-ben nem szereplő kódérték lesz. Ezután az Y betűvel kezdődően folytatjuk a bemeneti szöveg olvasását. Az algoritmus kicsit pontosabb megfogalmazásához legyen z egy szó típusú változó, K egy betű típusú változó. A z változó értéke kezdetben az összenyomni szánt állomány első betűje. Az eljárás futása során mindig teljesül, hogy $z \in S$. Az általános lépés a következő:

(0) Olvassuk a bemenő állomány következő betűjét K-ba.
(1) Ha az előző olvasási kísérlet sikertelen volt (vége a bemenetnek), akkor írjuk ki $c(z)$-t, és álljunk meg.
(2) Ha a zK szó is S-ben van, akkor $z \leftarrow zK$, és menjünk vissza (0)-ra.
(3) Különben (azaz ha $zK \notin S$) írjuk ki $c(z)$-t, tegyük a zK szót S-be. Legyen $z \leftarrow K$, majd menjünk vissza (0)-ra.

A módszer S-beli szavak egymásutánjára bontja a bemeneti szöveget. A felbontásban szereplő következő szónak mindig a leghosszabb lehetséges S-beli szót választja. Ez az ötlet, amit *mohó elemzésnek* neveznek, számos más szövegkezelő eljárásban is használható.

Ahogy korábban említettük, a $c(x)$ kódok rögzített hosszúságúak. Ha például ez a hosszúság 12 bit, akkor az S szótárba összesen 4096 szó kerülhet. Mi történjék akkor, ha a kódolás során kimerül ez a keret? Ha a bemeneti szöveg elég hosszú, akkor ez könnyen megtörténhet. Ebben az esetben a Lempel-Ziv-Welch-algoritmus felhagy a szótár további bővítésével. A (3) lépésben a zK szó S-be illesztését előíró rész többé nem hajtódik végre. A mohó elemzés a „betelt" S szótárra alapozva folyik tovább.

Kövessük az algoritmus működését egy példán. Az egyszerűség kedvéért legyen $\Sigma = \{a, b, c\}$. Kezdetben ezek az egybetűs szavak alkotják a szótárat. A szavak kódjai pozitív egészek legyenek; $c(a) = 1$, $c(b) = 2$, $c(b) = 3$, és az S-be kerülő új szavakhoz mindig a legkisebb még nem használt pozitív egészet rendeljük kódként. Legyen a bemenő szöveg $ababcbababaaaaaaa$. Kezdetben z értéke a. A K első értéke b lesz. Mivel $zK = ab$ nincs S-ben, először kiírjuk az eredmény első jeleként a $c(a) = 1$-et, az ab szót S-be illesztjük, és hozzá a négyest rendeljük kódként. A z értéke a lépés befejeztével b lesz. A következő lépésben $K = a$. Mivel $zK = ba$ nincs S-ben, az előzőhöz hasonlóan járunk el. Kiírjuk $c(b) = 2$-t, ba az S-be kerül, és a kódja 5 lesz, végül pedig $z = a$. Az általános lépés következő végrehajtásakor kerül először (2)-re a vezérlés, mivel ekkor már $ab \in S$. A következő input betű olvasása után $K = c$, $z = ab$. Mivel $zK = abc \notin S$, kiírjuk ab kódját, ami 4, az abc szót S-be illesztjük. A szó kódja 6 lesz, és végül $z = c$. Innen a folytatást az olvasóra bízzuk. A mohó elemzés által kapott felbontás $a|b|ab|c|ba|bab|a|aa|aaa|a$, a szöveg kódja pedig 1 2 4 3 5 8 1 10 11 1 lesz. Az S végső helyzetét és a benne levő szavak kódjait tartalmazza a következő táblázat:

a	1
b	2
c	3
ab	4
ba	5
abc	6
cb	7
bab	8
$baba$	9
aa	10
aaa	11
$aaaa$	12

A dekódolás, az összenyomott szöveg visszafejtése is elég egyszerű, de a kódoló eljárásnál azért valamivel bonyolultabb. A dekódolás első pillantásra ördöngösnek tetsző vonása, hogy *elvégezhető pusztán az egybetűs szavak kódjainak, valamint a kódok képzési szabályának ismeretében*. Nem kell tehát az egész S-et és a kódtáblát tárolni illetve elküldeni. Nem ismertetjük itt a teljes eljárást, csak az ötlet érzékeltetésére szorítkozunk. A lényeges mozzanat az, hogy a kódolt szöveget olvasva az egyes rész-szavak visszafejtése mellett rekonstruáljuk az S halmazt az elemeihez tartozó kódértékekkel együtt. Lássuk, hogyan boldogulunk az előbb kapott 1 2 4 3 5 8 1 10 11 1 sorozattal! Az első két jel betű kódja, ezeket hivatalból ismerjük. A szöveg első két betűje tehát ab. Ez a szó nem volt a kezdeti S-ben, de az eleje igen; innen rájövünk, hogy a kódoló algoritmus ezt $c(ab) = 4$ értékkel S-be tette. Ezt tudva a harmadik jel is megfejthető, így már a bemeneti szöveg első négy betűjét ismerjük: $abab$. Ebből a szövegdarabkából a kódoló algoritmus „fejével gondolkodva" kiderül, hogy ba volt a következő szó, ami S-be került, és ezáltal $c(ba) = 5$. Ebben a stílusban haladunk tovább. Legtöbbször a már megismert kódok közül kerül ki a következő visszafejtendő szám. Ennek megfejtése után pedig a bemeneti szöveg egy hosszabb szeletét ismerjük meg, amivel pedig az S szótárat építhetjük tovább.

Egyetlen eset van, amikor látszólag elakadunk, amikor a megfejtendő kód nem lesz az eddig kibontott szövegrész kódolása során kapott kódok között. Az összenyomást végző algoritmus ismeretében ekkor is tovább tudunk lépni. A helyzetet egy példával illusztráljuk. Tegyük fel, hogy az előbbi 1 2 4 3 5 8 1 10 11 1 sorozatot szeretnénk kibontani, és az ötöst már feldolgoztuk. Az eddigi kibontott szöveg $ababcba$, amin a kódoló munkáját utánozva felépíthetjük az S szótárat egészen a hetes kódértékű cb szóig. Ekkor látszólag bajban vagyunk, hiszen nyolcas kódú szó még nincs S-ben. Mi lehet ez az x szó? Az összenyomó módszer működését ismerve megállapíthatjuk, hogy az x szó $ba*$ alakú, ahol $*$ egy betű. Indoklásul

emlékeztetünk arra, hogy a kódoló algoritmus a mohó elemzés során utoljára leválasztott S-beli szót (ami esetünkben az ötös kódú ba) egészíti ki egy betűvel, és ezt teszi S-be.

Az x elejéből ezután kiderül az eredeti szöveg további két betűje: az ismert rész $ababcbaba$ lesz, amiből látjuk, hogy a $*$ betű csak b lehet, vagyis $x = bab$. A megfejtést tehát ekkor is folytatni tudjuk.

A Lempel–Ziv–Welch-módszer programja rövid és viharsebes. Mind a kódolás, mind pedig a dekódolás során csak egyszer kell olvasnunk a bemeneti jelsorozatot. A módszer gyorsabb a legjobb dinamikus Huffman-változatoknál, és erőteljesebb tömörítést is ér el azoknál a tipikus alkalmazások során (programszövegek, természetes nyelvű szövegek, grafika feldolgozása). A tömörítő képesség elemzése komoly statisztikai-információelméleti eszközöket igényel, így attól itt eltekintünk. Empirikus adatként megemlítjük, hogy a szokásos alkalmazásoknál $50-60\%$-ra teszik módszerrel elérhető hosszmegtakarítást. A Lempel–Ziv–Welch-algoritmust használja több ismert és népszerű tömörítő program, így például az újabb keletű UNIX változatok *compress* függvénye. Hasonló ötleteken alapulnak a széles körben használatos *zip* és *gzip* eljárások is.

6.

Gráfalgoritmusok

Kezdetben van a viszony.
Tekintsük a primitív népek nyelvét... E nyelv sejtmagjai – e nyelvtaniság előtti ősformák, melyek szétrobbanásából jön létre a szófajok sokfélesége, ezek általában valamely viszony egészét jelölik. Ahol mi azt mondjuk: „messze, távol", ott a zulu mondatszót mond, mely ennyit tesz: „Ott, ahol az ember ezt kiáltja: »Anyám, elvesztem!«." MARTIN BUBER: Én és Te

6.1. Bevezetés

Algoritmikus problémák megoldása során gyakran van szükségünk dolgok közötti bináris kapcsolatok ábrázolására, kezelésére. Erre természetes modellt kínálnak a gráfok. A gráf csúcsai az objektumoknak, az élei pedig a köztük levő (pár-)kapcsolatoknak felelnek meg. A probléma jellegéből adódóan használhatunk irányított vagy irányítatlan, súlyozott vagy súlyozatlan élű gráfokat. A gráf népszerű modellező eszköz egyszerűsége, kezelhetősége és kifejező képességének tágassága miatt. A seregnyi alkalmazási terület között előkelő helyen említhető maga a számítástudomány.

Az egyik legjelentősebb adatmodellező irányzat a hálós megközelítés[1], ami a leírni kívánt jelenségkört irányított gráfokkal ábrázolja. A modell alapfogalmai a gráf csúcsai, az élek pedig a köztük meglevő viszonyokat, kölcsönhatásokat, kapcsolatokat jelölik. Gráf az egész világ – sugallják a hálós filozófia hívei. Gráfoknak tekinthetők a belső memóriás adatkezelés (olykor igen szövevényes) listaszerkezetei is, ahol az adatelemek közötti mutatók (elterjedt szakkifejezéssel: pointe-

[1] A régebbi adatbázis-rendszerek közül ezt a modellt használja pl. az IDMS, az újabbak közül pedig a dbVISTA.

rek) jelentik az éleket. Ez a terület a gráfalgoritmusok időrendben első komoly alkalmazási terepe. Ebben a környezetben gyakran van szükség olyan algoritmusokra, amelyek elérhetőséggel, összefüggőséggel kapcsolatos kérdéseket válaszolnak meg, vagy hatékonyan bejárják a szerkezetet. Utóbbi feladat egy változatával, nevezetesen a bináris fák bejárásaival (pre-, in-, postorder) már találkoztunk.

Hasznosságukon túl az igazán csiszolt elemi gráfalgoritmusok sajátosan érdekesek is. Ennek a titka talán abban van, hogy itt három ismeretkör szálai fonódnak egybe: gráfokkal kapcsolatos matematikai tények (pl. a feszítőfák tulajdonságai), nagy erejű adatszerkezetek (pl. a kupac vagy az itt bemutatásra kerülő UNIÓ-HOLVAN) és általános algoritmikus elvek (pl. a mohó módszer vagy az optimalitás elve). A fejezetben tárgyalt módszerek tehát a közvetlen hasznukon túl egyszersmind az adatszerkezetekkel kapcsolatos ismeretek alkalmazásaként is szolgálnak, és előkészítő anyagot adnak a későbbi, általános algoritmuselméleti részekhez.

A fejezetben a meglehetősen szerteágazó anyagból csak néhány fontos és egyszerű, az alkalmazásokban gyakran felmerülő feladat és gráfalgoritmus ismertetésére szorítkozhatunk. Ezek az alapvető módszerek számos más, összetettebb algoritmus részét képezik, vagy éppen vázául szolgálnak. Némelyek közülük olyan gondolatot rejtenek magukban, amelyek kiindulópontjai voltak/lehetnek más problémák megoldásának.

Itt a bevezetőben tisztázzuk azokat a fogalmakat, jelöléseket, amelyeket a későbbiekben használni fogunk, és bemutatjuk a gráfok legtöbbször használt számítógépes ábrázolási módjait. Utána a legrövidebb utak keresésének feladatát tanulmányozzuk. Másik központi témánkat a hatékony bejáró algoritmusok (mélységi és szélességi bejárás) jelentik. Ezt követően a legismertebb feszítőfa-algoritmusokat tekintjük át. A fejezet végén a kombinatorikus optimalizálás irányába mutató feladatokat veszünk szemügyre: párosítások keresésével és hálózati folyamokkal foglalkozunk.

6.1.1. Alapfogalmak, jelölések

Egy G gráf két halmazból áll: a *csúcsok* vagy *pontok* V halmazából, mely egy véges, nem üres halmaz; és az *élek* E halmazából, melynek elemei bizonyos V-beli párok. Ha ezekről a pontpárokról kikötjük, hogy rendezettek legyenek, akkor *irányított gráfról*, különben pedig irányítatlan gráfról, vagy egyszerűen csak *gráfról* beszélünk. Az irányított gráfokat tehát nem szimmetrikus, míg az irányítatlanokat szimmetrikus kapcsolatok leírására használhatjuk. Már itt megjegyezzük, hogy egy irányítatlan gráf felfogható speciális irányított gráfként, ha minden élét két – oda és vissza mutató – irányított éllel helyettesítjük. Ezt az átírást használva

több irányított gráfokra megfogalmazott feladat (és azt megoldó algoritmus) értelmezhető (használható) irányítatlan gráfokra is. Fordítva, az irányított gráfok is tekinthetők irányítatlannak úgy, hogy elfeledkezünk az élek irányításról. Ekkor nem teszünk különbséget az (u, v) és a (v, u) pár (él) között.

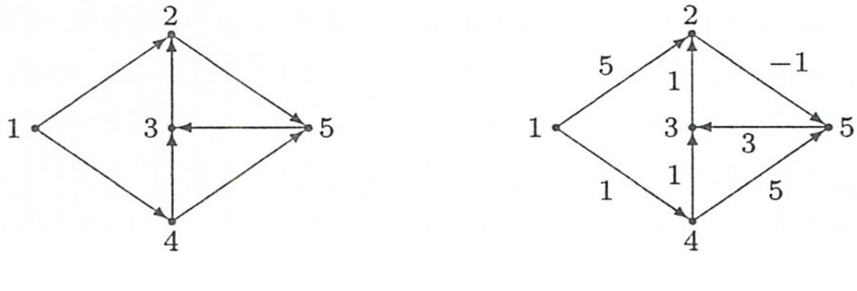
A H halmaz elemszámát $|H|$ jelöli. Az egyszerűség kedvéért az n és e betűk általában az adott gráf csúcs-, illetve élhalmazának elemszámát jelölik. Nevezetesen ha a $G = (V, E)$ gráfról beszélünk, akkor $n = |V|$ és $e = |E|$.

Az objektumok közötti kapcsolat sokszor jelenti út létezését vagy kommunikáció lehetőségét. Ilyenkor gyakran van szükség arra, hogy az élekhez *súlyokat* vagy *költségeket* tartsunk nyilván, melyek például időt, hosszúságot, pénzt és még sok mást ábrázolhatnak. Ezt általában egy valós értékű függvényként fogjuk fel, melynek értelmezési tartománya a gráf élhalmaza, és melyet c-vel, a *cost* angol szó kezdőbetűjével jelölünk.

Egy *út* – akár irányított, akár irányítatlan gráfban – csúcsok olyan $v_1, \ldots, v_k$ sorozata (ez lehet egyelemű is), melyre (v_i, v_{i+1}) minden i-re $(1 \leq i \leq k-1)$ éle a gráfnak. Egy utat *körnek* nevezünk, ha kezdő és végpontja megegyezik. Egy út, illetve egy kör *egyszerű*, ha minden csúcs, amin áthalad, különböző, kivéve a kör kezdő- és végpontját. Irányított gráfoknál a (v, w) él jelölésére használni fogjuk a $v \rightarrow w$ változatot is. Egy a v csúcsból a w-be menő irányított útra általában a $v \rightsquigarrow w$ jelöléssel utalunk.

Legyen $G = (V, E)$ egy – akár irányított, akár irányítatlan – gráf. Értelmezzük a $\sim$ relációt a V ponthalmazon a következőképpen: az $x, y \in V$ csúcsokra $x \sim y$, ha x és y között megy út (irányított gráfok esetén az irányítást nem figyelembe véve, azaz a $v \rightarrow w$ élen mehetünk w-ből v-be is). Könnyen belátható, hogy $\sim$ egy ekvivalenciareláció. A $\sim$ ekvivalenciaosztályait hívjuk a gráf *komponenseinek*. Egy gráf *összefüggő*, ha egyetlen komponensből áll. Egy gráfot *erdőnek* nevezünk, ha irányítatlan gráfként szemlélve nem tartalmaz kettőnél több pontból álló egyszerű kört (körmentesség). Ha egy erdőnek csak egyetlen komponense van, azaz összefüggő, akkor *fa*. Egy (irányított) gráf egy pontjának a *foka* a belőle kiinduló élek száma.

Példa: A következő ábrán egy irányított gráf és annak egy súlyozott élű változata látható. A gráf csúcshalmaza $V = \{1, 2, 3, 4, 5\}$, az élhalmaza pedig $E = \{(1,2), (1,4), (2,5), (3,2), (4,3), (4,5), (5,3)\}$. Tehát $n = 5$, és $e = 7$. A 4 csúcs foka 2.



A súlyozott változatnál az élekre írtuk azok költségeit. Így például a $2 \to 5$ él költsége $c(2,5) = -1$. Az 14325 pontsorozat egy $1 \rightsquigarrow 5$ irányított út; ez egyszerű út, mert a sorozatban nincs ismétlődés. Az 5325 sorozat pedig egy egyszerű irányított kör a gráfban.

A gráf összefüggő, hiszen ha az élek mentén a fordított irányban is léphetünk, akkor bárhonnan eljuthatunk bárhová. (Ha csak az élek irányát követve haladhatunk, akkor 1-be nem jutunk el egyetlen más csúcsból sem.)

Ha a gráfból töröljük a 2-nél nagyobb súlyú éleket, akkor a megmaradó G' gráf egy fa lesz. A G' csúcshalmaza V, az élei pedig $(1,4)$, $(4,3)$, $(3,2)$ és $(2,5)$. A G' gráf összefüggő és körmentes.

6.1.2. Gráfok ábrázolásai

A gráfok kényelmesen leírhatók számítógépes adatszerkezetekkel. Itt két széles körben használatos megadási módjukat ismertetjük, az adjacencia-mátrixot és az éllistát.

Adjacencia-mátrix

A $G = (V, E)$ gráf *adjacencia-mátrixa* (vagy *szomszédossági mátrixa*) a következő – a V elemeivel indexelt – n-szer n-es mátrix:

$$A[i,j] = \begin{cases} 0 & \text{ha } (i,j) \notin E, \\ 1 & \text{ha } (i,j) \in E. \end{cases}$$

Irányítatlan gráfok esetén a szomszédossági mátrix szimmetrikus lesz (azaz $A[i,j] = A[j,i]$ teljesül minden i, j csúcspárra). Ha az élekhez még költségeket is nyilván kell tartanunk, akkor az alábbi, árnyaltabb szerkezetű szomszédossági mátrixot fogjuk használni:

$$C[i,j] = \begin{cases} 0 & \text{ha } i = j, \\ c(i,j) & \text{ha } i \neq j \text{ és } (i,j) \text{ éle } G\text{-nek}, \\ * & \text{különben}. \end{cases}$$

Itt a $*$ szerepe csak annyi, hogy jelzi egy él nemlétét. Bizonyos esetekben más, az alkalmazás természetéhez jobban illeszkedő megkülönböztető jelet érdemes használni. Például később, a távolságok számításakor $*$ helyett a ∞ jelet használjuk, és „végtelenül nagy" súlyként értelmezzük. A főátlóba nullákat írtunk; ezt azért tettük, mert a súlyozott élű gráfokra vonatkozó feladatokban, amikkel foglalkozni fogunk, az $u \to u$ alakú ún. hurokélek érdektelenek.

Az adjacencia-mátrix természetes módon tekinthető kétdimenziós tömbnek. Ezzel kényelmes lehetőséget kínál gráfok számítógépes leírására.

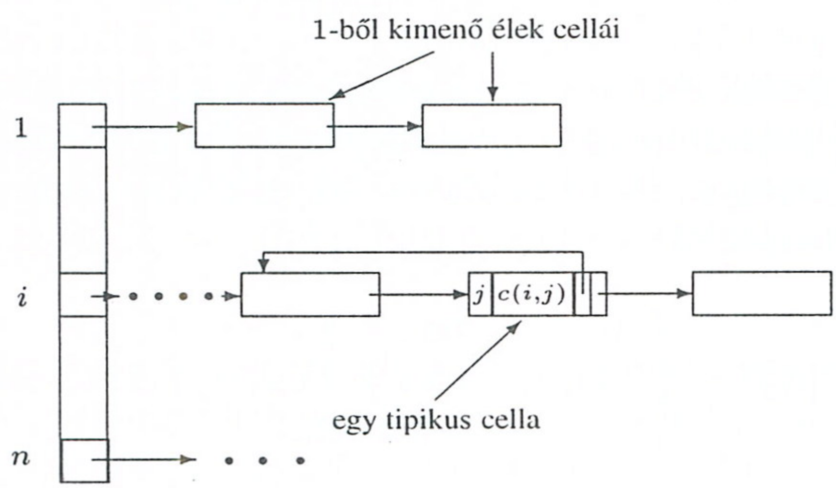
Az előző példa kétféle (egyszerű, illetve költségeket tartalmazó) adjacencia-mátrixa így fest:

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}, \quad C = \begin{pmatrix} 0 & 5 & * & 1 & * \\ * & 0 & * & * & -1 \\ * & 1 & 0 & * & * \\ * & * & 1 & 0 & 5 \\ * & * & 3 & * & 0 \end{pmatrix}.$$

Az adjacencia-mátrixszal való leírás egyszerű, programozástechnikai szempontból tiszta megoldás. Hátránya viszont, hogy a mérete (n^2 tömbelem) teljesen független az élek számától. Ha G-nek viszonylag kevés éle van, akkor gyakran hasznosabb az éllistás megadás.

Éllistás megadás

Ennél az ábrázolási módnál a $G = (V, E)$ gráf minden csúcsához egy lista tartozik. Az $i \in V$ csúcs listájában tároljuk az i-ből kimenő éleket, és ha kell, ezek súlyát is. Az i listáján egy élnek a lista egy eleme (cellája) felel meg. A lista elemeit a szokásos módon, mutatókkal láncoljuk egymás után (esetleg kétszeresen, visszafelé menő mutatókat is alkalmazva). Alább egy ilyen szervezés vázlata látható:



A (függőlegesen rajzolt) n méretű $L[1 : n]$ tömb $L[i]$ eleme az i csúcs éllistájának a feje. Az $L[i]$ tehát egy mutató az i-ből induló élek listájának első cellájára. Az (i, j) élnek megfelelő cella tartalmazza a j sorszámot, a $c(i, j)$ súlyt (ha van), egy mutatót a következő cellára, és esetleg még egyet az előzőre is.

Egy irányított gráf éllistás megadásában az n listán összesen e cella szerepel. Így az egész szerkezet elfér $n + e$ cellányi helyen. Irányítatlan gráfoknál a helyigény $n + 2e$ cella, hiszen az (i, j) él az i és a j csúcs listáján is megtalálható.

Ritka gráfok esetén, amikor e jóval kisebb, mint n^2, az éllistás megadás tömörebb az előzőnél. Az éllistás ábrázolás előnye még, hogy gyorsan végignézhetjük az egy csúcsból kiinduló összes élet. Az adjacencia-mátrixos megadás mellett viszont gyorsabban el tudjuk dönteni, hogy egy adott (i, j) pár éle-e G-nek.

6.2. A legrövidebb utak problémája (egy forrásból)

Legyen adott egy $G = (V, E)$ irányított gráf a $c(f)$, $f \in E$ élsúlyokkal. A csúcsok például jelölhetik egy többé-kevésbé hóval borított hegység különböző pontjait, az élek azt, hogy el lehet-e síelni egy sípályán az egyik pontból a másikba; az élsúlyok pedig azt, hogy mennyi idő kell ehhez. Felmerülhet a kérdés, hogy mekkora (itt időben mérve) a legrövidebb út *egy adott pontból egy másik adott pontba* vagy *egy adott pontból az összes többibe* vagy *bármely két pont között*. A meglepő az, hogy az első két probléma „ugyanolyan nehéz": az ismert algoritmusok, amelyek az elsőt megoldják, egyben a másodikat is megoldják. Tehát rögtön a második kérdéssel fogunk foglalkozni, a harmadik problémát pedig a következő részben tárgyaljuk.

Először fogalmazzuk meg pontosan a feladatot. A G gráf egy u-t v-vel összekötő (nem feltétlenül egyszerű) $u \rightsquigarrow v$ irányított útjának a *hossza* az úton szereplő élek súlyainak összege. *Legrövidebb* $u \rightsquigarrow v$ úton egy olyan $u \rightsquigarrow v$ utat értünk, amelynek a hossza minimális a G-beli $u \rightsquigarrow v$ utak között. Ezek után értelmezhetjük az u és v csúcsok (G-beli) $d(u, v)$ *távolságát*: ez 0, ha $u = v$; ∞, ha nincs $u \rightsquigarrow v$ út; egyébként pedig a legrövidebb $u \rightsquigarrow v$ út hossza. (Vigyázat, itt u és v nem felcserélhető: ha az egyik csúcs valamilyen távol van a másiktól, akkor nem biztos, hogy a másik is ugyanolyan távol van az egyiktől!) A meghatározás nem mindig értelmes. Ha például u és v egy olyan irányított körön vannak, amelynek az összsúlya negatív, akkor ezen körözve akármilyen kicsi (azaz nagy abszolút értékű negatív) úthosszat elérhetünk. Természetes kikötés tehát, hogy G ne tartalmazzon negatív összsúlyú irányított kört.

Jelöljünk ki a G gráfban egy $s \in V$ csúcsot forrásnak. Első célunk, hogy az $u \in V$ pontoknak az s-től való távolságát meghatározzuk. Tegyük fel továbbá, hogy a $c(f)$ élsúlyok nemnegatívak. Ekkor nyilván nincs az előbbi értelemben vett negatív kör.

A legrövidebb utak problémája (egy forrásból):

Adott egy $G = (V, E)$ irányított gráf, a $c : E \to \mathbb{R}^+$ nemnegatív értékű súlyfüggvény, és egy $s \in V$ csúcs (a forrás). Határozzuk meg minden $v \in V$-re a $d(s, v)$ távolságot.

6.2.1. Dijkstra módszere

A következőkben ismertetjük E. W. Dijkstra hatékony, ugyanakkor bűbájosan egyszerű algoritmusát az előbb megfogalmazott feladat megoldására. Egy a G csúcsaival indexelt $D[\,]$ tömböt használunk. A $D[v]$ értékre úgy gondolhatunk, hogy az minden időpillanatban az eljárás során addig megismert legrövidebb $s \rightsquigarrow v$ utak hosszát tartalmazza. A $D[v]$ mennyiség mindenkor felső közelítése lesz a keresett $d(s, v)$ távolságnak. A közelítést lépésről lépésre finomítva végül a kívánt értékeket kapjuk. Az algoritmus pontosabb megfogalmazásához először tegyük fel, hogy a G gráf az alábbi alakú C adjacencia-mátrixával adott:

$$C[v, w] = \begin{cases} 0 & \text{ha } v = w, \\ c(v, w) & \text{ha } v \neq w \text{ és } (v, w) \text{ éle } G\text{-nek}, \\ \infty & \text{különben.} \end{cases}$$

Kezdetben $D[v] := C[s, v]$ minden $v \in V$ csúcsra. Válasszuk ki ezután az s csúcs szomszédai közül a hozzá legközelebbit, vagyis egy olyan $x \in V \setminus \{s\}$ csúcsot, melyre $D[x]$ $(= C[s, x])$ minimális. Ekkor biztos, hogy az egyetlen (s, x) élből álló út egy legrövidebb $s \rightsquigarrow x$ út, hiszen bármerre másfele indulnánk el s-ből, már az első él miatt legalább ilyen hosszú utat kapnánk (az élsúlyok nemnegatívak!). Tehát x-et betehetjük (s mellé) a KÉSZ halmazba. A KÉSZ halmaz azokat a csúcsokat tartalmazza, amelyeknek s-től való távolságát már tudjuk. Ezek után módosítsuk a többi csúcs $D[w]$ értékét, ha az eddig ismertnél rövidebb úton el lehet érni oda x-en keresztül, azaz ha $D[x] + c(x, w) < D[w]$. Most újra válasszunk ki a $v \in V \setminus$ KÉSZ csúcsok közül egy olyat, amelyre $D[v]$ minimális. Ezen csúcs $D[\,]$-értéke már az s-től való távolságát tartalmazza az előzőhöz hasonló okok miatt (ezt később bebizonyítjuk). Majd megint a $D[\,]$-értékeket módosítjuk, és így tovább, míg minden csúcs be nem kerül a KÉSZ halmazba.

Az eljárás egy *mohó algoritmus*: a KÉSZ halmaz bővítésekor látszólag nem a teljes kép figyelembe vételével választ optimumot. Mégis ezek a helyinek, korlátozott érvényűnek tűnő döntések együtt elvezetnek a tényleges optimumokhoz. A módszert ezek után formálisan a következő programvázlat írja le:

Dijkstra algoritmusa adjacencia-mátrixszal

(1) KÉSZ := $\{s\}$
 for minden $v \in V$ csúcsra **do**
 $D[v] := C[s, v]$ (∗ a $d(s, v)$ távolság első közelítése ∗)
(2) **for** $i := 1$ **to** $n - 1$ **do begin**
 Válasszunk olyan $x \in V \setminus$ KÉSZ csúcsot, melyre $D[x]$ minimális.
 Tegyük x-et a KÉSZ-be.
 for minden $w \in V \setminus$ KÉSZ csúcsra **do**
(3) $D[w] := \min\{D[w], D[x] + C[x, w]\}$ (∗ $d(s, w)$ új közelítése ∗)
 end

A módszer helyességének igazolásához hasznos lesz a *különleges út* fogalma: egy $s \rightsquigarrow z$ irányított út különleges, ha a z végpontot kivéve minden pontja a KÉSZ halmazban van. A különleges úttal elérhető pontok éppen a KÉSZ-ből egyetlen éllel elérhető pontok.

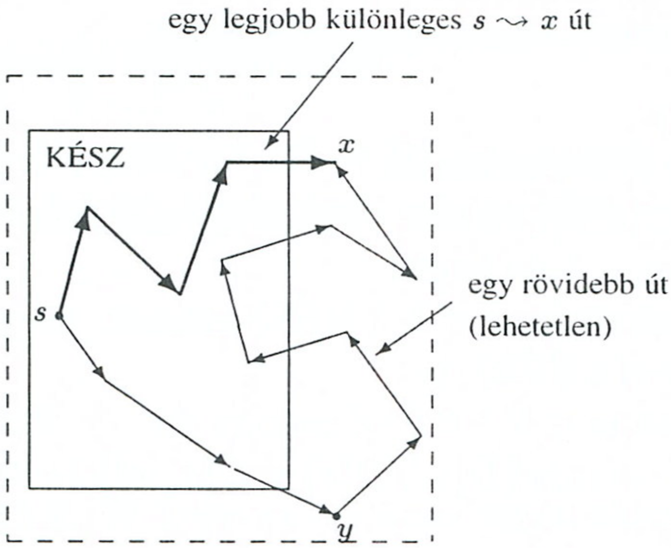
Állítás: *A (2) ciklus minden iterációs lépése után érvényesek a következők:*

(a) KÉSZ pontjaira $D[v]$ a legrövidebb $s \rightsquigarrow v$ utak hossza.

(b) Ha $v \in$ KÉSZ, akkor van olyan $d(s, v)$ hosszúságú (más szóval legrövidebb) $s \rightsquigarrow v$ út is, amelynek minden pontja a KÉSZ halmazban van.

(c) Külső (vagyis $w \in V \setminus$ KÉSZ) pontokra $D[w]$ a legrövidebb különleges $s \rightsquigarrow w$ utak hossza.

Bizonyítás: Teljes indukciót alkalmazunk. A kezdőértékek beállítása után, amikor a (2) ciklus még egyszer sem futott le, mindhárom állítás nyilvánvalóan igaz. Tegyük fel, hogy igazak a j-edik iteráció után, azaz miután a (2) ciklus magja j-szer lefutott. Azt szeretnénk belátni, hogy igazak maradnak a $j + 1$-edik iteráció után is. Tegyük fel, hogy az algoritmus a $j + 1$. iterációs lépésben az x csúcsot választja a KÉSZ-be.
(a) Gondolkozzunk indirekte: mi van, ha $D[x]$ nem a $d(s, x)$ távolságot jelöli, azaz van ennél rövidebb $s \rightsquigarrow x$ út? Ez nem lehet különleges, elvégre a (c)-re vonatkozó indukciós feltevés szerint $D[x]$ az x-be vezető legrövidebb különleges utak hosszát tartalmazza. Ezen út „eleje" azonban különleges, mert a KÉSZ-ből indulva egy azon kívüli csúcsba jut el. Legyen y az út első olyan pontja, amely már nincs benne a KÉSZ halmazban. Ekkor az $s \rightsquigarrow y$ útdarab hossza is kisebb, mint $D[x]$. Az útdarab különleges, így a hossza nem kevesebb, mint $D[y]$. (Itt ismét használtuk a (c)-re vonatkozó indukciós feltevést.) Ezeket összevetve kiderül, hogy a j-edik iteráció után $D[y] < D[x]$, ami képtelenség, hiszen ekkor az algoritmusnak y-t kellett volna választania x helyett.



(b) Ezt is elég csupán az x csúcsra igazolni, hiszen a KÉSZ korábbi pontjaira az indukciós feltevés adja az állítást. Már beláttuk, hogy $d(s,x) = D[x]$. Az utóbbi érték egy különleges $s \rightsquigarrow x$ út hossza volt a $j+1$. iteráció előtt (itt a (c)-re vonatkozó indukciós feltevést használtuk); annak végeztével az út minden pontja KÉSZ-beli lesz.

(c) A (b) alapján $d(s,v) + C[v,w]$ a legrövidebb olyan $s \rightsquigarrow w$ különleges utak hossza, amelyek utolsó előtti pontja v. Emiatt az állítás egyenértékű a

$$D[w] = \min_{v \in \text{KÉSZ}} \{d(s,v) + C[v,w]\},$$

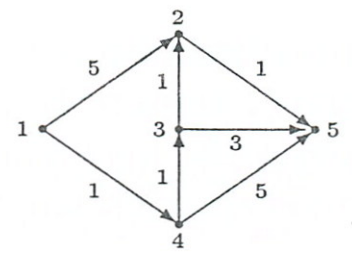
egyenlőséggel. Miért lesz ez igaz? A (c)-re vonatkozó indukciós feltevés alapján a $D[w]$ régi (a $j+1$. iteráció előtti) értéke

$$D[w] = \min_{v \in \text{KÉSZ} \setminus \{x\}} \{d(s,v) + C[v,w]\}.$$

Ezt – a már igazolt $d(s,x) = D[x]$ egyenlőséget is figyelembe véve – a (3) sorban a $d(s,x) + C[x,w]$ mennyiséggel vetjük össze. A $D[w]$-be tehát tényleg az új KÉSZ halmazra vett minimum kerül. □

Végül amikor KÉSZ már a teljes csúcshalmaz, $D[v] = d(s,v)$ teljesül minden $v \in V$ csúcsra az állítás (a) része szerint. Ezzel igazoltuk az algoritmus helyességét. Most vizsgáljuk meg az időigényét (n csúcs esetén). A kezdőértékek beállítása $O(n)$ elemi lépést vesz igénybe. A (2) ciklus belsejében a minimumkeresés és a (3) ciklus szintén $O(n)$-et, így mivel a (2) ciklus $n-1$-szer fut le, *az algoritmus összidőigénye $O(n^2)$ művelet.*

Példa: Nézzük a Dijkstra-algoritmus működését a következő rajzon látható G gráffal mint bemenettel; a forrás az $s := 1$ csúcs:



A D tömb helyzetét mutatjuk a (2) iteráció megkezdése előtt, majd pedig az első három iterációs menet után. Az utolsó, a negyedik menetben csak a KÉSZ halmaz változik, D nem. Vastag betűvel szedtük a KÉSZ-beli csúcsokhoz tartozó értékeket. Ezek a megfelelő legrövidebb utak hosszai. Érdemes megfigyelni a $D[5]$ alakulását. Látható, hogy mindhárom menetben csökkent, míg végül elérte a $d(1,5)$ értéket.

1	2	3	4	5
0	5	∞	1	∞

1	2	3	4	5
0	5	2	**1**	6

1	2	3	4	5
0	3	**2**	**1**	5

1	2	3	4	5
0	**3**	**2**	**1**	4

Dijkstra algoritmusa éllistával

Ha a G gráf ritka, vagyis kevés éle van, akkor az időigény csökkenthető, amennyiben a gráfot éllistával tároljuk. Az előző algoritmust a következőképpen módosítjuk: $V \backslash$KÉSZ csúcsait *kupacba* rendezve tartjuk a $D[\]$ érték szerint. A kezdeti kupacépítés $O(n)$ költséggel végezhető el. A (2) ciklus belsejében levő minimumkeresést egy $O(\log n)$ költségű MINTÖR művelettel oldjuk meg. A $D[\]$ értékek újraszámolását és a kupac-tulajdonság helyreállítását (utóbbi csúcsonként egy $O(\log n)$ költségű FOGYASZT) csak a választott csúcs szomszédaira kell elvégezni. Minden csúcsot pontosan egyszer választunk ki, és a szomszédok számának összege e. *Tehát itt az összidőigény* $O((n+e)\log n)$. Ez figyelemre méltó javítás a mátrixos megoldáshoz képest, ha a gráf ritka, azaz e sokkal kisebb, mint n^2.

Sűrű gráfokra még komolyabb gyorsítás érhető el alkalmas d-kupaccal. Ekkor az előző bekezdésben taglalt teendők költsége $O(n + nd\log_d n + e\log_d n)$ lesz. Itt az első tag a kupacépítés, a második a minimumok, a harmadik a fogyasztások lépésszáma. Tegyük fel, hogy G-nek sok éle van, mondjuk $n^{1,5} \le e \le n^2$ és legyen $d = \lceil e/n \rceil$. Ekkor $d \ge \sqrt{n}$, amiből $\log_d n \le 2$. Ezt használva

$$O(n + nd\log_d n + e\log_d n) = O(n + nd + e) = O(n + n \cdot e/n + e) = O(e).$$

Az algoritmus ebben az esetben a bemenet hosszával arányos lépésszám alatt végez – más szóval: *lineáris idejű*. Ebből következően elmondhatjuk, hogy sűrű gráfokra ez az implementáció konstans szorzótól eltekintve optimális. A módszer eme

változata ékesszólóan példázza az adatszerkezetek alkalmazásaiban rejlő lehetőségeket. Az egyszerű kupacot (vagyis 2-kupacot) használó megoldáshoz képest azért nyertünk, mert sokkal több a fogyasztás, mint a minimumok törlése. Az előbbiek pedig olcsóbbak, ha nagy a kupac d elágazási tényezője.

Nyilvánvaló, hogy a Dijkstra-algoritmus irányítatlan gráfokra is működik, hiszen egy irányítatlan gráf felfogható olyan irányított gráfnak, melyben minden él mindkét irányba mutat. Megjegyzésképp annyit, hogy irányítatlan gráfok esetében az élsúlyokra vonatkozó kétféle kikötés ugyanazt jelenti (vagyis hogy ne legyen negatív összsúlyú kör, illetve hogy az élsúlyok nemnegatívak), hiszen egy negatív irányítatlan élből egy (kettő hosszúságú) negatív kört kapunk.

A legrövidebb utak nyomonkövetése

Sokszor persze nemcsak a legrövidebb utak hosszára, hanem magukra a legrövidebb utakra is kíváncsiak vagyunk. Mindkét algoritmust a futási idő nagyságrendjének megtartása mellett kibővíthetjük egy $P[\,]$ tömb karbantartásával, amely minden csúcshoz megadja egy az eddig ismert hozzá vezető legrövidebb úton az utolsó előtti csúcsot. Azért elég csupán az utolsó előtti csúcsot tárolni, mert ha egy v csúcsba vezető legrövidebb útból elhagyjuk az utolsó élet, akkor nyilván az utolsó előtti csúcsba vezető legrövidebb utat kapunk, aminek szintén ismerjük a cél előtti állomását, és így tovább. Visszafelé felgombolyíthatunk egy a v-be vivő legrövidebb utat.

Tehát kezdetben $P[v] := s$ minden $v \in V$-re. Ezután csak annyit kell hozzátenni az algoritmushoz, hogy a (3) ciklus belsejében, ha egy külső w csúcs $D[w]$ értékét megváltoztatjuk, akkor $P[w] := x$, hiszen ekkor találtunk egy rövidebb speciális utat w-be, melynek utolsó előtti állomása x. Könnyű meggondolni, hogy végül tényleg igaz lesz az, hogy $P[v]$ egy legrövidebb $s \rightsquigarrow v$ út utolsó előtti pontja; másként fogalmazva: $d(s, v) = d(s, P[v]) + C[P[v], v]$ és $P[v] \neq v$ érvényes lesz minden $v \in V \setminus \{s\}$-re.

Feladat: Mutassuk meg, hogy a $P[v] \to v$ élek egy fát alkotnak. A fa $P[v] \to v$ élének a súlya legyen $C[P[v], v]$. Ekkor a fában az s-től mért távolságok ugyanazok lesznek, mint a G-beli távolságok.

6.2.2. A Bellman—Ford-módszer

Olyan módszert ismertetünk az egy pontból induló legrövidebb utak (hosszának) meghatározására, amely akkor is működik, ha bizonyos élsúlyok negatívak. Csupán annyit teszünk fel, hogy *G nem tartalmaz negatív összhosszúságú irányított*

kört. Mint látni fogjuk, a nagyobb általánosságért idővel fizetünk. Az eljárás n^3-bel arányos számú műveletet igényel.

Feladat: Mutassuk meg, hogy ha a súlyozott élű G irányított gráfban nincs negatív összhosszúságú irányított kör, akkor bármely két pontja között van olyan legrövidebb út is, ami egyszerű (azaz nem tartalmaz ismétlődő csúcsot). Következésképpen van nem több, mint $n-1$ élből álló legrövidebb út is, ahol n a G csúcsainak száma.

Tegyük fel itt is, hogy a $G = (V, E)$ súlyozott élű irányított gráf a C adjacencia-mátrixával adott (ebben a diagonális elemek nullák, az i, j helyzetű elem a $c(i, j)$ élsúly, ha $i \to j$ éle G-nek, a többi elem pedig ∞). Az egyszerűség kedvéért tegyük még fel, hogy $V = \{1, 2, \ldots, n\}$ és $s = 1$. A szakirodalomban többnyire R. E. Bellmannak és L. R. Fordnak tulajdonított algoritmus fokozatos közelítéssel határozza meg az s-ből a többi csúcsba vivő legrövidebb utak hosszát.

A módszer tulajdonképpen egy $T[1 : n-1, 1 : n]$ táblázat (kétdimenziós tömb) sorról sorra haladó kitöltése. Azt szeretnénk elérni, hogy végül minden i, j-re ($1 \leq i \leq n-1$, $1 \leq j \leq n$) teljesüljön, hogy

$$(*) \qquad T[i, j] = \begin{array}{l}\text{a legrövidebb olyan } 1 \rightsquigarrow j \text{ irányított utak hossza,}\\ \text{melyek legfeljebb } i \text{ élből állnak.}\end{array}$$

A feladatban foglalt állítás szerint ekkor $T[n-1, j]$ a legrövidebb $1 \rightsquigarrow j$ utak hosszát tartalmazza. A módszert úgy tekinthetjük, hogy a keresett minimális távolságokat $n-1$ menetben közelítjük. Az első menet, a $T[1, j]$ sor kitöltése kézenfekvő, hiszen nyilván $T[1, j] = C[1, j]$. Tegyük fel ezután, hogy az i-edik sort már kitöltöttük, azaz a $T[i, 1], T[i, 2], \ldots, T[i, n]$ értékekre $(*)$ igaz. Ekkor

$$(**) \qquad T[i+1, j] := \min\{T[i, j], \min_{k \neq j}\{T[i, k] + C[k, j]\}\}$$

adja az $i+1$. sor helyes értékeit. Ugyanis egy legfeljebb $i+1$ élből álló $\pi = 1 \rightsquigarrow j$ út kétféle lehet:

(a) Az útnak kevesebb, mint $i+1$ éle van. Ekkor ennek a hossza szerepel $T[i, j]$-ben.

(b) Az út éppen $i+1$ élből áll. Legyen l a π út utolsó előtti pontja. Ekkor a π út $1 \rightsquigarrow l$ szakasza i élből áll, és a π minimalitása miatt minimális hosszúságú a legfeljebb i élű $1 \rightsquigarrow l$ utak között. A hossza tehát a T már elkészült darabjára tett feltevésünk miatt $T[i, l]$. Eszerint a π hossza $T[i, l] + C[l, j]$.

A fejtegetés tanulsága, hogy a π hossza szerepel a $(**)$ jobboldalán azon mennyiségek között, amelyek minimumát vesszük; tehát egyenlő a minimummal.

A (**) rekurzió fontos és hasznos vonása, hogy a $T[i+1, j]$ értéket adó minimális utak kezdődarabját, az $1 \rightsquigarrow l$ részt (illetve annak hosszát) nem kell hatalmas

szénakazalban keresnünk. Tudjuk ugyanis, hogy ez az útdarab is rendelkezik egy optimalitási tulajdonsággal: minimális hosszú a legfeljebb i élből álló $1 \leadsto l$ utak között. Optimalizálási feladatoknál gyakran találkozunk hasonló helyzettel, amikor is az optimumot adó megoldás – alkalmasan definiált – részeinek is rendelkezniük kell valamiféle optimalitási tulajdonsággal. Ez pedig rendszerint azért igaz, mert ha a részeken tudnánk javítani, akkor az javítást adna globális értelemben is. A jelenségre úgy szokás hivatkozni, hogy teljesül az *optimalitás elve*.

Nézzük ezután a módszer költségét! A T táblázat egy értékének a $(**)$ szerinti számítása $n-1$ összeadást és ugyanennyi összehasonlítást (minimumkeresés n elem közül) igényel. Mindez összesen $O(n^3)$ elemi műveletet jelent.

A T táblázat kitöltésével egyidőben gondoskodhatunk a minimális utak nyomonkövetéséről is. A Dijkstra eljárásánál alkalmazott megoldáshoz hasonlóan elegendő feljegyezni minden j csúcsra egy legrövidebb $1 \leadsto j$ út utolsó előtti pontját. Ezt egy $P[1:n]$ tömb alkalmazásával oldhatjuk meg. A $P[j]$ értéke legyen az eljárás futása során az addig ismert legrövidebb $1 \leadsto j$ út utolsó előtti pontja (kezdetben $P[j]=1$ minden j-re). A részletek kimunkálását az olvasóra hagyjuk.

Végezetül megemlítjük, hogy a módszer mintegy $3n$ tárcella alkalmazásával is megvalósítható – szemben a kb. n^2-tel, amit az itt leírt változat használ.

6.3. Floyd módszere az összes csúcspár közötti távolság meghatározására

Ebben a részben először azt a problémát tárgyaljuk, hogy miként lehet egy irányított gráfban az összes pontpár távolságát meghatározni, majd ennek a problémának egy speciális esetét, amikor csak arra vagyunk kíváncsiak, hogy mely pontok között vezet egyáltalán irányított út. Végül pedig az első algoritmus egy alkalmazását mutatjuk be.

Tehát nézzük először az általános kérdést! Nemnegatív élsúlyok esetén nyilvánvaló, hogy ha a Dijkstra-algoritmust minden csúcsra mint forrásra lefuttatjuk, akkor az összes rendezett pontpár távolságát megkapjuk. Van azonban egy ennél közvetlenebb módszer, amelynek előnye, hogy akkor is működik, ha van a gráfban negatív élsúly (természetesen negatív összsúlyú irányított kör nem lehet). Ez a módszer R. W. Floyd nevéhez fűződik.

A probléma

Adott egy $G=(V,E)$ irányított gráf, és egy $c: E \to \mathbb{R}$ súlyfüggvény úgy, hogy a gráfban nincs negatív összsúlyú irányított kör. Határozzuk meg az összes $v, w \in V$ rendezett pontpárra a $d(v,w)$ távolságot.

6.3.1. Floyd módszere

Tegyük fel továbbra is, hogy $V = \{1, 2, ..., n\}$, és hogy a G gráf a C adjacenciamátrixával adott. A pontpárok távolságának kiszámítására egy szintén $n \times n$-es F mátrixot fogunk használni. Kezdetben $F[i, j] := C[i, j]$. Ezután egy ciklust hajtunk végre, amelynek k-adik lefutása után $F[i, j]$ azon $i \rightsquigarrow j$ utak legrövidebbjeinek a hosszát tartalmazza, amelyek közbülső pontjai k-nál nem nagyobb sorszámúak.

Az új $F_k[i, j]$ értékeket a következőképpen számíthatjuk ki a $k - 1$-edik iteráció utáni $F_{k-1}[i, j]$ értékekből (itt az index csak F különböző pillanatbeli értékeire utal, és nem ennyi különböző mátrixra). Nyilván egy legrövidebb $i \rightsquigarrow j$ út, melyen a közbülső pontok sorszáma legfeljebb k, vagy tartalmazza a k csúcsot vagy nem. Ha igen, akkor ezen út első fele egy olyan legrövidebb $i \rightsquigarrow k$ út, ami mentén a közbülső pontok sorszáma legfeljebb $k - 1$, tehát hosszát $F_{k-1}[i, k]$ már tartalmazza; második fele meg szintén egy ilyen tulajdonságú legrövidebb $k \rightsquigarrow j$ út, aminek a hossza $F_{k-1}[k, j]$. Használtuk itt, hogy a legrövidebb utak között vannak egyszerűek is (lásd a Feladatot a Bellman–Ford-nál), és itt is segítségünkre volt az optimalitás elve.[2]

Ha az út nem tartalmazza k-t, akkor $F_k[i, j] = F_{k-1}[i, j]$. Mivel $F_k[i, k] = F_{k-1}[i, k]$ és $F_k[k, j] = F_{k-1}[k, j]$, az algoritmust végrehajthatjuk az F mátrix egyetlen példányával.

FLOYD algoritmusa

(1) **for** $i := 1$ **to** n **do**
 for $j := 1$ **to** n **do**
 $F[i, j] := C[i, j]$
(2) **for** $k := 1$ **to** n **do**
 for $i := 1$ **to** n **do**
 for $j := 1$ **to** n **do**
 $F[i, j] := \min\{F[i, j], F[i, k] + F[k, j]\}$

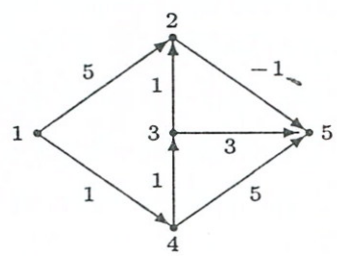
A megelőző gondolatmenet alapján k szerinti indukcióval azonnal adódik a következő, a Floyd-módszer „ciklusinvariánsát" megfogalmazó állítás.

Állítás: *$F[i, j]$ a (2)-beli iteráció k-adik menete után azon legrövidebb $i \rightsquigarrow j$ utak hosszát tartalmazza, amelyek belső csúcsai $1, 2, \ldots, k$ közül valók.* □

[2] A Bellman–Ford- és a Floyd-módszer a *dinamikus programozás* elnevezésű általános algoritmikus megközelítés jegyeit viseli. A dinamikus programozással később külön is foglalkozunk.

Az állítást a $k = n$ esetre alkalmazva következik a módszer helyessége, vagyis hogy végül $F[i, j] = d(i, j)$ igaz minden i, j csúcspárra. Az algoritmus lépésszáma n^3-bel arányos, hiszen a domináló rész a három egymásba ágyazott **for** ciklus. Ez nagyságrendben ugyanannyi, mintha a Dijkstra mátrixos változatát minden csúcsra mint forrásra lefuttatnánk (pontosabb elemzéssel kiderül, hogy a Floyd-módszer mintegy 50%-kal gyorsabb). Ha a gráf éleinek száma jóval kisebb n^2-nél, és az élsúlyok nemnegatívak, akkor érdemesebb a Dijkstra-algoritmus éllistás változatát használni.

Példa: Nézzük, mihez kezd a Floyd-algoritmus a következő súlyozott élű G gráffal. Látható, hogy G-ben nincs negatív kör.



Az F tömböt négyzetes mátrixként ábrázoljuk. F_i jelentése: az F tömb i-edik iterációs lépés utáni állapota. Az utolsó mátrix a páronkénti legrövidebb utak hosszát tartalmazza.

$$F_0 = F_1 = \begin{pmatrix} 0 & 5 & \infty & 1 & \infty \\ \infty & 0 & \infty & \infty & -1 \\ \infty & 1 & 0 & \infty & 3 \\ \infty & \infty & 1 & 0 & 5 \\ \infty & \infty & \infty & \infty & 0 \end{pmatrix}, \quad F_2 = \begin{pmatrix} 0 & 5 & \infty & 1 & 4 \\ \infty & 0 & \infty & \infty & -1 \\ \infty & 1 & 0 & \infty & 0 \\ \infty & \infty & 1 & 0 & 5 \\ \infty & \infty & \infty & \infty & 0 \end{pmatrix},$$

$$F_3 = \begin{pmatrix} 0 & 5 & \infty & 1 & 4 \\ \infty & 0 & \infty & \infty & -1 \\ \infty & 1 & 0 & \infty & 0 \\ \infty & 2 & 1 & 0 & 1 \\ \infty & \infty & \infty & \infty & 0 \end{pmatrix}, \quad F_4 = F_5 = \begin{pmatrix} 0 & 3 & 2 & 1 & 2 \\ \infty & 0 & \infty & \infty & -1 \\ \infty & 1 & 0 & \infty & 0 \\ \infty & 2 & 1 & 0 & 1 \\ \infty & \infty & \infty & \infty & 0 \end{pmatrix}.$$

Az $F_0 = F_1$ és $F_4 = F_5$ egyenlőségek azért igazak, mert sem 1, sem 5 nem lehet egy út közbülső csúcsa, hiszen 1-be nem fut él, és 5-ből nem indul ki él. Ugyanezekkel magyarázható a sok ∞ érték az első oszlopban és az utolsó sorban.

A legrövidebb utak nyomonkövetése

A legrövidebb utak nyomonkövetése megint csak egy menet közben karbantartott – itt már $n \times n$-es – P tömb segítségével történhet. Kezdetben $P[i,j] := 0$. Az algoritmushoz annyit kell hozzátenni, hogy ha a (2) ciklus legbelsejében egy $F[i,j]$ értéket megváltoztatunk, mert találtunk egy k-n átmenő rövidebb utat, akkor $P[i,j] := k$. Végül $P[i,j]$ egy legrövidebb $i \rightsquigarrow j$ út „középső" csúcsát fogja tartalmazni. Az $i \rightsquigarrow j$ út összeállításához tehát találnunk kell egy i-ből ebbe a közbülső csúcsba vezető legrövidebb utat, és egy ebből a csúcsból j-be vezető legrövidebb utat. De ezen utaknak is ismerjük egy-egy „középső" csúcsát, és így tovább. A következő program a P tömb segítségével kiír egy legrövidebb $i \rightsquigarrow j$ utat.

procedure legrövidebb út(i, j:csúcs);
var k:csúcs;
begin
 $k := P[i,j]$;
 if $k = 0$ **then return**;
 legrövidebb út(i, k);
 kiír(k);
 legrövidebb út(k, j)
end;

Feladat: Mutassuk meg, hogy az előző kiíró program bizonyos P mátrixok esetén végtelen ciklusba kerülhet, de a Floyd-algoritmus eredményeként kapott P mátrixszal nem.

6.3.2. Tranzitív lezárás

A bemenet ismét egy $G = (V, E)$ irányított gráf. Most nem a legrövidebb utakra, hanem csak arra vagyunk kíváncsiak, hogy mely pontok között vezet irányított út. Persze használhatjuk Floyd módszerét is, hiszen ha az algoritmus lefutása után $F[i,j]$ értéke véges, akkor van $i \rightsquigarrow j$ út, ha pedig végtelen, akkor nincs. Kicsit egyszerűbb algoritmust kapunk, ha a G gráf A adjacencia-mátrixát (amiben csak 0 és 1 értékek szerepelnek) Boole-mátrixként fogjuk fel, és így a műveletek logikai műveletek lesznek. Ez az algoritmus megelőzte Floydét, és S. Warshall nevéhez fűződik.

Definíció (tranzitív lezárt): *Legyen $G = (V, E)$ egy irányított gráf, A az adjacencia-mátrixa. Legyen továbbá T a következő $n \times n$-es mátrix:*

$$T[i,j] = \begin{cases} 1 & \textit{ha } i\textit{-ből } j \textit{ elérhető irányított úttal;} \\ 0 & \textit{különben.} \end{cases}$$

Ekkor a T mátrixot – illetve az általa meghatározott gráfot – az A mátrix – illetve az általa meghatározott G gráf – (reflexív) tranzitív lezártjának *hívjuk.*

A probléma

Adott a (Boole-mátrixként értelmezett) A adjacencia-mátrixával a $G = (V, E)$ irányított gráf. Adjuk meg a G tranzitív lezártját.

Warshall algoritmusa tranzitív lezárásra

Floyd algoritmusában csak a következő változtatásokat kell tennünk. Az (1) ciklusban a kezdőértékek beállítása helyett

$$T[i,j] := \begin{cases} 1 & \text{ha } i = j \text{ vagy } A[i,j] = 1, \\ 0 & \text{különben.} \end{cases}$$

A (2) ciklusban F értékeinek változtatása helyett (ugyanazt megfogalmazva logikai műveletekkel)

$$(+) \qquad T[i,j] := T[i,j] \vee (T[i,k] \wedge T[k,j]).$$

A Floyd-algoritmussal kapcsolatos megállapításaink megfelelői itt is érvényesek. Jelesül a k-adik iteráció után $T[i,j] = 1$, azaz *igaz*, ha van olyan $i \leadsto j$ út, amelynek belső pontjai k-nál nem nagyobbak; és 0, azaz *hamis*, ha ilyen út nem létezik. A (+) értékadás a (2) ciklus belsejében azt fejezi ki, hogy ilyen $i \leadsto j$ út akkor létezik, ha

1. már van olyan $i \leadsto j$ út, amely nem megy keresztül $k-1$-nél nagyobb sorszámú csúcson, vagy
2. vannak olyan $i \leadsto k$ és $k \leadsto j$ utak, amelyek nem mennek keresztül $k - 1$-nél nagyobb sorszámú csúcson.

Az előző részhez hasonlóan itt is fenntarthatunk egy P tömböt, amiből gyorsan kiolvashatunk egy $i \leadsto j$ utat, ha ilyen létezik G-ben. Az algoritmus időigénye nyilvánvalóan $O(n^3)$.

6.3.3. Egy alkalmazás: centrum keresése irányított gráfban

A legrövidebb utak témájától elköszönőben a Floyd-módszer egyik alkalmazását mutatjuk be. Először is tisztázzuk, hogy mit értünk egy irányított gráf centrumán. Ezt egy analógiával készítjük elő.

Egy körlap minden x pontjához hozzárendelhetjük a körlap többi pontjától való távolságának $e(x)$ maximumát. Ezt nyilván úgy kaphatjuk meg, hogy meghúzzuk az x-en átmenő átmérőt. Az x két szakaszra osztja az átmérőt, és az $e(x)$ érték ezen szakaszok közül a nem rövidebbnek a hossza. A kör középpontja az a pont, melyre az $e(x)$ érték minimális. Ezzel párhuzamban értelmezzük egy gráf centrumát:

A súlyozott élű G irányított gráf v csúcsára legyen

$$e(v) = \max\{d(w,v) : \quad w \in V\}.$$

A v csúcs *centruma* G-nek, ha $e(v)$ minimális az összes $v \in V$ között.

Algoritmus centrum keresésére:

(1) Először Floyd módszerével kiszámítjuk a G-beli pontpárok közötti távolságokat. Ez $O(n^3)$ műveletet jelent.

(2) Az előző lépésben kapott F mátrix minden oszlopában meghatározzuk a maximális értéket (azaz kiszámítjuk az $e(v)$ értékeket). Itt n-szer keresünk maximumot n elem közül; ennek költsége $O(n^2)$.

(3) Végül megkeressük az n darab $e(v)$ érték minimumát. Ennek a műveletigénye $O(n)$.

A (3)-beli minimumot adó (bármelyik) v csúcs a G centruma lesz. A számítás legköltségesebb része az (1) lépés, így a műveletigény nagyságrendje $O(n^3)$.

6.4. Mélységi bejárás

Gráfokkal kapcsolatos algoritmikus feladatokban gyakran van szükségünk arra, hogy az élek mentén lépdelve valamilyen szisztematikus módon végigjárjuk a gráf csúcsait. Az erre a célra való *bejáró algoritmusok* központi jelentőségűek a gráfalgoritmusok között. Bejáró módszerek adják a gráf-struktúrákon működő összetett algoritmusok vezérlési szerkezetét azzal, hogy szabályozzák a sorrendet, amely szerint a csúcsokhoz kötődő munkákat elvégezzük. A bejáró módszerek különösen alkalmasak gráfok szerkezetének feltérképezésére; segítségükkel pontról pontra lépkedve összegyűjthetjük a kívánt információkat. Bináris fák bejárásaival (pre-, in- és postorder) már foglalkoztunk. Itt két általános, tetszőleges gráfra működő módszert ismerünk meg. Ezek a *mélységi bejárás* vagy *keresés* (depth-first-search, DFS) és a *szélességi bejárás* vagy *keresés* (breadth-first-search, BFS). Mindkettő számtalan gráfalgoritmus alapvázául szolgál.

Hogy szemléletes képet kapjunk a két stratégiáról, nézzük meg az öreg városka girbe-gurba utcáin bolyongó kopott emlékezetű lámpagyújtogató esetét. Minden

este szürkületkor elindul, hogy végigjárja a városka lámpáit (a gráf pontjai), de sosem tudja hol jár, csak arra emlékszik, hogy merről jött, és egy-egy lámpánál az onnan induló utcácskákba (a gráf élei) bepillantva látja, hogy arrafelé a következő lámpa ég-e már. Tehát ahol épp éri az alkonyat, felgyújtja az útjába kerülő első lámpát, majd elindul az egyik – még nyilván sötét – utcán. Felgyújtja a következőt is, majd továbbmegy arra, ahonnan nem lát fényt. Ezt addig folytatja, míg egy olyan lámpához nem ér, ahonnan kiinduló összes utcából már fény dereng feléje. Ekkor visszamegy arra, amerről ide érkezett, és onnan próbál másfele, a még meg nem gyújtott lámpák irányába indulni. Ha már erre sincs ilyen utca, akkor még visszább megy. Egy idő után visszaérkezik a kiindulóhelyére, és megpróbál másfele elindulni. Ha már minden irányt végigjárt, újra visszajut a kezdőpontra, ahol mindenfelől lámpák fénye pislákol feléje, úgyhogy nyugodtan megy aludni. Módszerének lényege, hogy addig ment egyre „mélyebben" a városkába, ameddig csak volt olyan hely, ahol még nem járt.

A másik módszer szemléltetésére képzeljük el, hogy az egyik este a kopott emlékezetű lámpagyújtogató elhívja az összes kopott emlékezetű barátját – akik meglehetősen sokan vannak –, hogy segítsenek neki a lámpagyújtogatásban. A szenilis barátok – miután az első lámpát közösen meggyújtották – annyifelé oszlanak, ahány utca indul onnan. Meggyújtják az összes szomszédos lámpát, majd tovább osztódnak, és indulnak a még sötét utcák felé. Tehát így „széltében" terjeszkedve özönlik el a várost.

Hogy lássuk a két stratégia közötti lényeges különbséget, nézzük felülről a várost. Az első módszer szerint haladva a kiindulóponttól viszonylag távoli helyeken már akkor is lesz fény, amikor még a kezdőpont környékén van sötét lámpa. A második viszont a kezdőpont kis környezetében gyújt először teljes világosságot.

Ez a példa rámutat arra is, hogy a szélességi bejárás hatékonyan párhuzamosítható, míg a mélységi bejárásra hasonló megoldást nem ismerünk. Természetesen az előbbinek is van párhuzamosságot nem használó implementációja, ahol a szomszédos pontokat nem egyszerre, hanem valamilyen sorrendben látogatjuk meg.

A szélességi bejárás megoldja a legrövidebb utak problémáját egy forrásból abban a speciális esetben, amikor minden él súlya egy. Először feltérképezi a forrástól egységnyi távolságra levő csúcsokat, majd minden további menetben az eggyel távolabb levőket.

Ebben a részben a mélységi bejárást tárgyaljuk részletesebben, néhány hozzá kapcsolódó fogalommal, algoritmussal és alkalmazással együtt.

6.4.1. Irányított gráfok mélységi bejárása

A mélységi bejárás egyfajta mohó menetelés. Egy kezdőpontból kiindulva addig megyünk tovább irányított élek mentén, ameddig már nem tudunk még be nem

járt csúcsba jutni. Ez esetben visszamegyünk az út utolsó előtti pontjáig, és onnan próbálkozunk másfele tovább lépni és előremenni. Ha ezek a lehetőségek is kimerültek, még eggyel visszamegyünk, és így tovább. Azt, hogy a v pontból már nem tudunk előre lépni, úgy fejezzük ki, hogy a v pont mélységi bejárása befejeződött. Előfordulhat, hogy már a kezdőpont bejárását is befejeztük, de a gráfnak még mindig van olyan csúcsa, ahol nem jártunk. Ez azt jelenti, hogy a kezdőpontunkból nem érhető el minden pont irányított úton. Ekkor egy új kezdőpontot választunk a még be nem járt csúcsok közül, és innen folytatjuk a gráf mélységi bejárását.

Legyen $G = (V, E)$ egy irányított gráf, ahol $V = \{1, \ldots, n\}$. Jelölje $L[v]$ a v csúcs éllistáját ($1 \leq v \leq n$). Vegyünk föl továbbá egy bejárva$[1 : n]$ logikai vektort, amely minden ponthoz megadja, hogy jártunk-e már ott. A G gráf mélységi bejárását a következő programmal valósíthatjuk meg:

```
procedure bejár (* elvégzi a G irányított gráf mélységi bejárását *)
    begin
        for v := 1 to n do
            bejárva[v] := hamis;
        for v := 1 to n do
            if bejárva[v] = hamis then
              mb(v)
    end
```

```
procedure mb (v: csúcs)
    var
        w: csúcs;
    begin
(1)     bejárva[v] := igaz; (* meggyújtjuk a lámpát *)
        for minden L[v]-beli w csúcsra do
(2)         if bejárva[w] = hamis then
              mb(w) (* megyünk a következő még sötét lámpához *)
    end
```

Minden csúcsra pontosan egyszer hívjuk meg az *mb* eljárást, és ott egy csúcs minden szomszédját pontosan egyszer vizsgáljuk meg. Ezért az algoritmus időigénye $O(n + e)$, vagy ami ugyanaz, $O(\max(n, e))$. Gondoljuk meg, hogy $e + n$ lépés pusztán a gráf beolvasásához is szükséges. Tehát ennél lényegesen gyorsabb bejáró eljárás nem képzelhető el. A módszer *lineáris idejű*, amin azt értjük, hogy a bemenet hosszával arányos időn belül végez.

A *bejár* eljárásban az esetleges új kezdőpont a legkisebb (sorszámú) még nem bejárt pont lesz. Ez a választás nem lényeges a módszer működése és időigénye szempontjából. A fontos csak annyi, hogy viszonylag kevés munkával találjunk új kiindulópontot.

Mélységi számok és befejezési számok

A mélységi bejárás során sok hasznos információt gyűjthetünk a gráfról. Ennek egyik módja, hogy a csúcsok mellé feljegyezzük az algoritmus néhány fontosabb történését. Itt kétféle ilyen eseménnyel foglalkozunk. A lámpás képet használva: rögzítjük azt az időpontot, amikor meggyújtjuk a lámpát, és azt is, amikor visszaérünk egy lámpához, aminek már minden szomszédja világít.

Mindezeket a *mélységi* és a *befejezési* számok segítségével valósítjuk meg. Felvesszük az mszám$[1 : n]$ és bszám$[1 : n]$ egész értékű vektorokat. Az mszám$[v]$ azt fogja megadni, hogy a v csúcsot a bejárás során hányadikként látogattuk meg. A bszám$[v]$ pedig azt fogja mutatni, hogy hányadikként fejeződött be a v csúcs mélységi bejárása.

Definíció (mélységi számozás): *A G irányított gráf csúcsainak egy* mélységi számozása *a gráf v csúcsához azt a sorszámot rendeli, mely megadja, hogy az mb eljárás (1) sorában a csúcsok közül hányadikként állítottuk bejárva$[v]$ értékét igaz-ra. A v csúcs mélységi számát mszám$[v]$ jelöli.*

Definíció (befejezési számozás): *A G irányított gráf csúcsainak egy* befejezési számozása *a gráf v csúcsához azt a sorszámot rendeli, mely megadja, hogy a csúcsok közül hányadikként ért véget az mb(v) hívás. A v csúcs befejezési számát bszám$[v]$ jelöli.*

A mélységi és befejezési számok meghatározásához a kódot a következőkkel egészítjük ki: legyenek *msz* és *bsz* egész értékű változók, melyekben a már kiadott legnagyobb mélységi, illetve befejezési sorszámokat szándékozzuk tárolni. A *bejár* eljárás elején beállítjuk a kezdőértékeket. Ez az

```
msz := 0;
bsz := 0;
```

utasításokkal tehető meg. Ugyanezen eljárás első **for** ciklusában az

```
mszám[v] := 0;
bszám[v] := 0;
```

értékadásokkal kinullázzuk a két új tömböt. Írjuk továbbá az *mb* eljárás (1) sora után az

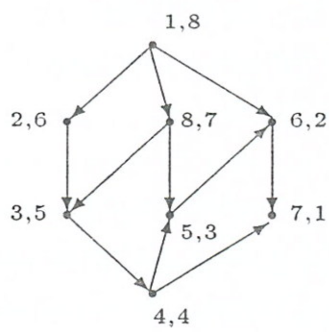
$$\text{msz} := \text{msz} + 1;$$
$$\text{mszám}[v] := \text{msz};$$

utasításpárt, és a (2) utasítás után (vagyis az **end** elé) a

$$\text{bsz} := \text{bsz} + 1;$$
$$\text{bszám}[v] := \text{bsz};$$

kódrészletet. A kiegészítésekkel nem növeljük meg lényegesen a futási időt, hiszen mindezek csúcsonként konstans mennyiségű extra munkát jelentenek. A módszer lépésszáma így is $O(n + e)$ marad.

Példa: A következő gráf csúcsai mellett feltüntettük az egyik lehetséges mélységi bejárása szerinti mélységi, illetve befejezési számokat. A kiindulópont a gráf legfelső csúcsa.



Feladat: Tegyük fel, hogy egy bináris fa mélységi bejárását végezzük a gyökértől indulva. Tegyük fel továbbá, hogy a fa élei lefelé, az apától a fiú felé irányítottak, és a csúcsok éllistáin mindig a bal fiú szerepel először (ha létezik egyáltalán). Mutassuk meg, hogy ekkor a csúcsok növekvő mélységi szám szerinti sorrendje éppen a preorder bejárás szerinti sorrend. Ugyanígy a csúcsok növő befejezési szám szerinti sorrendje megegyezik a postorder sorrenddel.

A gráf éleinek osztályozása és a mélységi feszítő erdő

A mélységi bejárás lehetővé teszi a gráf éleinek egy érdekes osztályozását. Hívjuk *faéleknek* azokat az éleket, melyek megvizsgálásukkor még be nem járt pontba mutatnak (azok az utcák, amikbe bepillantva még sötét van, ezért arra indulunk a következő lámpához).

Definíció (faél): *A $G = (V, E)$ irányított gráf $v \to w$ éle* faél *(az adott mélységi bejárásra vonatkozóan), ha megvizsgálásakor a (2) sorban a (bejárva$[w] =$ hamis) feltétel teljesül.*

Jelölje T azt a gráfot, amelynek csúcshalmaza V (vagyis a kiinduló G gráf csúcshalmaza), élei pedig a faélek. Az algoritmus szerkezetéből nyilvánvaló, hogy egy csúcsba legfeljebb egy faél futhat be (egy lámpát csak egyszer gyújtunk meg).

Feladat: Mutassuk meg, hogy ha egy H összefüggő irányított gráfnak van olyan csúcsa, amelybe nem vezet él, és az összes többi csúcsába pontosan egy él fut be, akkor H körmentes (nincs benne kettőnél hosszabb egyszerű irányítatlan kör).

A feladat állítása szerint T egy erdő. A komponenseit olyan fák alkotják, melynek pontjait egy kezdőpont bejárása során értük el. Ennek megfelelően egy ilyen fa élei a kezdőponttól – amit a fa *gyökerének* is szokás nevezni – távolodóan irányítottak.

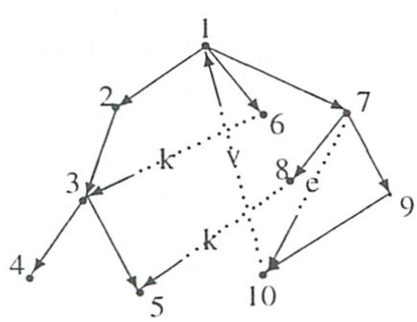
Definíció (mélységi feszítő erdő, feszítőfa): *Az előbb meghatározott T gráfot a G gráf egy* mélységi feszítő erdejének *nevezzük. Ha T csak egy komponensből áll, akkor* mélységi feszítőfáról *beszélünk.*

A gráf azon élei, melyek nem faélek, még további három csoportba sorolhatók. Ezt előkészítendő nevezzük a G gráf y csúcsát az x csúcs T-beli *leszármazottjának*, ha van $x \rightsquigarrow y$ irányított út T-ben. (Tehát x is leszármazottja önmagának.) Az x csúcs *részfáját* az x leszármazottai és a köztük menő faélek alkotják.

Definíció (élek osztályozása): *Tekintsük a G irányított gráf egy mélységi bejárását és a kapott T mélységi feszítő erdőt. (Ezen bejárás szerint) G egy $x \rightarrow y$ éle*

faél,	*ha $x \rightarrow y$ éle T-nek;*
előreél,	*ha $x \rightarrow y$ nem faél, y leszármazottja x-nek T-ben, és $x \neq y$;*
visszaél,	*ha x leszármazottja y-nak T-ben (a hurokél is ilyen);*
keresztél,	*ha x és y nem leszármazottai egymásnak.*

A következő ábrán egy gráf egy mélységi feszítőfáját, a csúcsok mélységi számozását és éleinek típusát láthatjuk:



Most nézzük, miként lehet hatékonyan felismerni az egyes éltípusokat. Egy v csúcs minden valódi leszármazottjának a mélységi száma nagyobb mszám$[v]$-nél, hiszen T élei mentén haladva a mélységi számok nőnek. Ezért egy előreél kisebb mélységi számú csúcsból nagyobb sorszámúba, egy visszaél pedig – a hurokélek kivételével – nagyobb sorszámú csúcsból kisebb sorszámúba mutat. Az ábra azt sugallja, hogy a keresztélek is nagyobb sorszámúból kisebb sorszámú csúcsba vezetnek. Valóban így is van: megmutatjuk, hogy ha $x \rightarrow y \in E$ és mszám$[x]$ <mszám$[y]$, akkor az $x \rightarrow y$ csak faél vagy előreél lehet. Ez esetben ugyanis az $mb(x)$ hívás időpontjában y-t még nem látogattuk meg. Az $x \rightarrow y$ élet mindenképpen megvizsgáljuk, mielőtt ez a hívás befejeződne. Ha a vizsgálat pillanatáig y-ban még nem jártunk, ezt az élet faélnek kell választanunk. Különben y az x-nek leszármazottja, ami azt jelenti, hogy az $x \rightarrow y$ él előreél.

Az eddigiek alapján – pusztán a mélységi számokat használva – a visszaéleket még nem tudjuk megkülönböztetni a keresztélektől. Segítségül hívjuk ezért a befejezési számokat. Ha az $x \rightarrow y$ él visszaél, vagyis x leszármazottja y-nak, akkor az x csúcsot nyilván y bejárása alatt látogatjuk meg, következésképpen az $x \rightarrow y$ él vizsgálatakor az y bejárása még nem fejeződött be. Ekkor tehát még bszám$[y] = 0$. Ha viszont $x \rightarrow y$ él keresztél, akkor x nincs benne az y részfájában. A már igazolt mszám$[y] <$ mszám$[x]$ egyenlőtlenség szerint y bejárása elkezdődött az x bejárása előtt. E két tényből következik, hogy y bejárása befejeződött, mielőtt x-et meglátogattuk. Az $x \rightarrow y$ él vizsgálatakor tehát már bszám$[y] > 0$.

Mindezek alapján a következő táblázat foglalja össze, hogy egy irányított gráf mélységi bejárása során hogyan tudjuk megkülönböztetni a négyféle éltípust:

$x \rightarrow y$ egy	ha az él vizsgálatakor
- faél	mszám$[y] = 0$
- visszaél	mszám$[y] \leq$ mszám$[x]$ és bszám$[y] = 0$
- előreél	mszám$[y] >$ mszám$[x]$
- keresztél	mszám$[y] <$ mszám$[x]$ és bszám$[y] > 0$.

A visszaéleknél egyenlőséget is megengedtünk. Ez azt jelenti, hogy az esetleges $x \rightarrow x$ hurokéleket visszaéleknek tekintjük. A táblázatban foglalt feltételek ellenőrzése állandó mennyiségű pluszmunkát ad élenként; az algoritmus ezzel kiegészítve is lineáris idejű marad:

Tétel: *A G irányított gráf mélységi bejárása – beleértve a mélységi, a befejezési számozást és az élek osztályozását is –* $O(n+e)$ *lépésben megtehető.* □.

A következő állítás a mélységi feszítő erdő részfáinak egy hasznos jellemzését adja. Tekintsük a $G = (V, E)$ irányított gráf egy T mélységi feszítő erdejét.

Legyen $x \in V$ egy tetszőleges csúcs, és jelölje T_x a feszítő erdő x-gyökerű részfájának a csúcshalmazát. A T_x elemei éppen az x leszármazottai a szóban forgó mélységi bejárásra vonatkozóan. Legyen továbbá

$$S_x = \left\{ y \in V \,\middle|\, \begin{array}{l} \text{van olyan } G\text{-beli } x \rightsquigarrow y \text{ irányított út, amelyen} \\ \text{a csúcsok mélységi száma legalább mszám}[x] \end{array} \right\}.$$

Az S_x halmaz tehát azokból a pontokból áll, amelyek x-ből elérhetők olyan $u \to v \in E$ élek mentén, melyek végpontjaira mszám$[u]$ $\geq$mszám$[x]$, és mszám$[v]$ $\geq$mszám$[x]$.

Állítás (részfa-lemma): *Tetszőleges $x \in V$ csúcs esetén érvényes a $T_x = S_x$ egyenlőség.*

Bizonyítás: T_x éppen azokból a pontokból áll, amelyek x-ből faélek mentén elérhetők. Az x-gyökerű részfa csúcsait x után érjük el a mélységi bejárás során, ezért a részfa bármely $u \to v$ élére mszám$[u]$ $\geq$mszám$[x]$, és mszám$[v]$ $\geq$mszám$[x]$. Ezzel beláttuk, hogy $T_x \subseteq S_x$.

A fordított irányú tartalmazás igazolására tegyük fel indirekte, hogy létezik egy $y \in S_x \setminus T_x$ csúcs. Legyen $x \rightsquigarrow y$ egy az S_x meghatározásában szereplő irányított út. Az $x \rightsquigarrow y$ út helyett esetleg annak egy kezdődarabját véve feltehetjük, hogy az út utolsó előtti v pontja T_x-ben van. A mélységi bejárás során x-szel kezdődően a T_x pontjait látogatjuk meg; a részfán kívüli pontokra ezért vagy x előtt, vagy T_x csúcsai után kerül sor. Az $y \in S_x$ feltétel szerint mszám$[y]$ $>$mszám$[x]$. Ez $y \notin T_x$ miatt azt jelenti, hogy y-t valamikor a T_x pontjai után látogatjuk meg, amiből arra jutunk, hogy mszám$[y]$ $>$mszám$[v]$. Ebből következik viszont, hogy $v \to y$ faél vagy előreél. Mindkét esetben $y \in T_x$ adódik, ellentmondva feltevésünknek. Ezzel beláttuk, hogy $S_x \subseteq T_x$ is igaz. □

Következmény: *Tegyük fel, hogy a $G = (V, E)$ gráf x csúcsából minden pont elérhető irányított úton. Tegyük fel továbbá, hogy a G mélységi bejárását x-szel kezdjük. Ekkor a mélységi feszítő erdő egyetlen fából áll.*

Bizonyítás: Ennél a bejárásnál mszám$[x] = 1$, amiből a feltétel alapján $S_x = V$. A részfa-lemma szerint tehát $T_x = V$. □

Feladat: Igazoljuk, hogy

$$T_x = \{y \in V \mid \text{mszám}[y] \geq \text{mszám}[x] \text{ és bszám}[y] \leq \text{bszám}[x]\}.$$

6.4.2. Irányított körmentes gráfok (DAG-ok)

Tennivalói közé tartozik a salétromozás utáni napon annak a pácnak a felvétele, amely marékszámra vett sóból, korianderből, borókamagból, sárga-cukorból, darált fokhagymából, babérlevélből és öt-hat fej vöröshagymából készült. KRÚDY GYULA: A húsvéti sódar titkai

Itt irányított gráfok egy olyan osztályával foglalkozunk, amely gyakran kerül elő a különféle alkalmazásokban. Másrészt arról is nevezetesek ezek a gráfok, hogy körükben néhány elemi algoritmikus feladat gyorsabban és egyszerűbben megoldható, mint általában.

Definíció (DAG): *Egy G irányított gráf* DAG (directed acyclic graph), *ha nem tartalmaz irányított kört.*

DAG-ok a tudomány és a tervezés számos területén felbukkannak. Gyakran hordoz valamiféle helyességgel, konzisztenciával kapcsolatos jelentést az, hogy a modellezésre használt irányított gráf DAG-e, vagy sem. Ízelítőül két példát mutatunk.

1. Teendők ütemezése. Képzeljük el, hogy egy összetett tevékenység a $T_1, \ldots, T_n$ részteendőkre bontható. Ezek általában nem függetlenek egymástól. Gyakran vannak olyan feltételeink, hogy bizonyos (i, j) párokra a T_j teendőt csak a T_i teendő elvégzése után hajthatjuk végre. Például ha a sódarkészítést egy összetett tevékenységnek fogjuk fel, akkor – Krúdy Gyula receptje szerint – a salétromozás előbb kell hogy legyen, mint a pácolás.

Az ilyen feltételrendszerek természetes módon leírhatók irányított gráffal. Ennek csúcsai a T_i teendők. A T_i-ből a T_j-be akkor vezet él, ha a T_i-t a T_j előtt kell elvégezni. A receptnél maradva: a salétromozás részteendőtől él vezet a pácoláshoz; a páckészítés bizonyos részfeladatai – mondjuk a koriander és a babérlevél hozzáadása – között pedig nem fut él egyik irányban sem.

Látni fogjuk később, hogy a teendőket akkor és csak akkor tudjuk a feltételeket kielégítő sorrendben végrehajtani, ha az itt vázolt gráf DAG. Az ilyen ütemezési gráfok élsúlyokkal gazdagított változatai a PERT-gráfok; ezekről még lesz szó a későbbiekben.

2. Várakozási gráfok. Temérdek olyan számítógépes rendszer van, ami egyidőben több felhasználót szolgál ki. Ilyennek tekinthető például egy légitársaság helyfoglalási adatbázisa. A tárolt adatokkal egyszerre több program – szokásos kifejezéssel: tranzakció – dolgozik. Hogy ne legyen nagy kavarodás (mondjuk hogy több

utas ugyanarra a helyre kap jegyet), egy tranzakció zárolja az adatbázis azon részeit, amelyekkel dolgozni kíván. A zár időtartama alatt más tranzakció nem férhet ezekhez. Ha a T_2 tranzakciónak egy olyan A adatra volna szüksége, amit a T_1 használ, és ezért zárol, akkor T_2 vár amíg A felszabadul. Vannak azonban olyan – pattnak nevezett – helyezetek, amikor a várakozás nem segít: amikor néhány tranzakció „körben vár egymásra", és ezért egyikük sem tudja folytatni a munkát. A tranzakciók munkáját felügyelő program a patthelyzet észlelése után kilövi az egyik érintett T-t, hogy a többiek tovább dolgozhassanak.

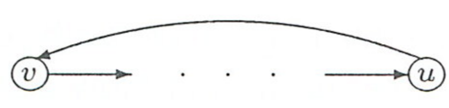
A pattok felismerésére szolgál a *várakozási gráf*. Ez egy irányított gráf, aminek csúcsai az aktív tranzakciók. A T_1 tranzakcióból él fut a T_2-be, ha van olyan A adat, amit T_2 használni (pl. írni vagy olvasni) szeretne, és amit T_1 éppen zár alatt tart. A tranzakcióink között pontosan akkor nincs patthelyzet, ha a várakozási gráf egy DAG. Ez közvetlenül adódik a topologikus rendezésről szóló tételből, amit mindjárt tárgyalni fogunk.

A DAG-tulajdonság ellenőrzése

Fontos tehát, hogy egy irányított gráfról el tudjuk dönteni, tartalmaz-e irányított kört. Ha a gráf egy mélységi bejárása során találunk visszaélet, akkor a gráf nyilván tartalmaz irányított kört, azaz nem DAG. Ugyanis az $u \to v$ visszaél v pontjából vezet (csupa faélekből álló) út u-ba. A következő egyszerű tétel azt mondja ki, hogy ez visszafele is igaz, így a gráf mélységi bejárása elegendő a DAG-tulajdonság ellenőrzéséhez.

Tétel: *Legyen $G = (V, E)$ egy irányított gráf. Ha G egy DAG, akkor egyetlen mélységi bejárása során sincs visszaél. Fordítva: ha G-nek van olyan mélységi bejárása, amelyre nézve nincs visszaél, akkor G egy DAG.*

Bizonyítás: Az első állítást az előbb már beláttuk. A második indoklásához tegyük fel, hogy G nem DAG. Meg kell mutatnunk, hogy tetszőleges mélységi bejárása során lesz visszaél. A G nem DAG, így tartalmaz legalább egy irányított kört. Vegyük ennek a körnek azt a v csúcsát, melynek a szóban forgó mélységi bejárás során a legkisebb a mélységi száma, és vizsgáljuk meg a kör v-be mutató élét. Ennek kezdőpontja legyen u.



Mivel mszám$[v]$ < mszám$[u]$, ez az él csak vissza- vagy keresztél lehet. Azonban v-ből u elérhető irányított úton, nevezetesen a kör élein haladva. Ezen az úton a pontok mélységi száma nem kisebb, mint mszám$[v]$. Alkalmazható a részfa-lemma: u a v leszármazottja lesz a mélységi feszítő erdőben. Így az $u \to v$ él csak visszaél lehet. □

Ezek után egy irányított gráfról mélységi bejárás segítségével $O(n+e)$ idő alatt el tudjuk dönteni, hogy DAG-e, vagy sem. Ha a bejárás közben találunk visszaélet, akkor a gráf nem DAG, különben pedig igen.

DAG topologikus rendezése

Emlékezzünk vissza a teendők ütemezésének problémájára. Adva van tehát n darab teendőnk, vagyis az ezeket reprezentáló n darab csúcs. Egy $u \to v$ él azt jelenti, hogy az u teendőt a v előtt kell elvégezni. Ilyenkor nyilván a teendők olyan sorrendjét keressük, ahol ha egy teendőt előbb kell elvégeznünk egy másiknál, akkor ez a sorban előbb van. Így a teendőket ebben a sorrendben elvégezhetjük anélkül, hogy megsértenénk az élekben megtestesülő feltételeket.

Definíció (topologikus rendezés): *Legyen $G = (V, E)$ ($|V| = n$) egy irányított gráf. G egy* topologikus rendezése *a csúcsoknak egy olyan $v_1, \ldots, v_n$ sorrendje, melyben $x \to y \in E$ esetén x előbb van, mint y (azaz ha $x = v_i, y = v_j$, akkor $i < j$).*

Tétel: *Egy irányított gráfnak akkor és csak akkor van topologikus rendezése, ha DAG.*

Bizonyítás: $\Rightarrow$: Ha G nem DAG, akkor nem lehet topologikus rendezése, mert egy irányított kör csúcsainak nyilván nincs megfelelő sorrendje.

$\Leftarrow$: Tegyük fel, hogy G egy DAG. Először megjegyezzük, hogy G-ben van olyan csúcs, amibe nem fut be él. Ha ugyanis minden pontba menne él, akkor egy tetszőleges pontból kiindulva lépkedni tudnánk visszafelé az irányított éleken anélkül, hogy valahol is elakadnánk. E séta során egyszer „körbeérnénk", ami ellentmond a DAG-tulajdonságnak.

Ezután a csúcsok száma szerinti teljes indukciót alkalmazunk. Ha G-nek csak egy pontja van, akkor az állítás nyilván igaz. Tegyük fel, hogy $n-1$-pontú DAG-ok esetén is teljesül, és legyen G egy tetszőleges n pontú DAG. G-ből hagyjunk el egy olyan x csúcsot a belőle kiinduló élekkel együtt, amibe nem fut be él. A kapott gráf legyen G_1. Nyilván G_1 is DAG. Az indukciós feltevés szerint a G_1-nek létezik $w_1, \ldots, w_{n-1}$ topologikus rendezése. Mármost az $x, w_1, \ldots, w_{n-1}$ sorrend nyilván a G egy topologikus rendezése. □

A teendők elvégezhetőségének tehát tényleg szükséges és elegendő feltétele, hogy a hozzájuk rendelt gráf DAG legyen. Hasonló mondható a tranzakciós példánkról: a várakozási gráf topologikus rendezhetőségéből következik, hogy valamelyik T biztosan megkaphatja a hőn áhított zárat.

A tétel gondolatmenetében egy DAG-ok topologikus rendezésére szolgáló algoritmus lappang: válasszunk ki első pontnak egy olyan csúcsot, amibe nem fut él, majd a maradékból egy ilyen csúcsot másodikként; és így tovább. Az ötlet a hatékonyan kivitelezhető a *sor* adatszerkezet segítségével. A sor alapja egy *elem*ekből álló kettősen láncolt lista. Két alapművelet van: sorba(v, Q) a v elemet a Q sor végére illeszti; első(Q) pedig visszaadja és egyben kitörli Q-ból annak első elemét. Nyilvánvaló, hogy ezek a műveletek a lista elejét, illetve végét jelölő mutatókkal $O(1)$ költséggel megvalósíthatók. A sort szokás még FIFO-listának (first in, first out) is nevezni.

Most egy Q sort használó topologikus rendező algoritmus vázlata következik.

1. A (kezdetben üres) Q sorba illesszük be a G azon csúcsait, amelyekbe nem megy él.
2. Ha Q üres, akkor álljunk meg, különben legyen $u :=$ első(Q), és írjuk ki az u csúcsot.
3. Töröljük a gráfból az $u \to v$ éleket. Ha egy itt előforduló v csúcsba már nem megy él, akkor sorba(v, Q).
4. Menjünk vissza a 2. lépéshez.

Feladat: Mutassuk meg, hogy az előző algoritmus kiírási sorrendje tényleg a G topologikus rendezése, és hogy az eljárás $O(n + e)$ költséggel megvalósítható.

Ugyancsak lineáris költségű topologikus rendező eljárást kaphatunk a mélységi bejárás segítségével: végezzük el a G DAG egy mélységi bejárását, és írjuk ki G csúcsait a befejezési számaik szerint növekvő $w_1, \ldots, w_n$ sorrendben. Ez megtehető az $O(n + e)$ költségkorlát megtartása mellett. A w csúcsot akkor írjuk ki, amikor a bejárása befejeződött, vagyis az mb(w) hívás utolsó utasításaként.

Állítás: *A $w_n, w_{n-1}, \ldots, w_1$ sorrend a G DAG egy topologikus rendezése.*

Bizonyítás: Azt kell belátnunk, hogy ha $w_i \to w_j$ éle G-nek, akkor $i > j$. Ez azon múlik, hogy G-ben nincs visszaél. A $w_i \to w_j$ él tehát vagy fa- vagy előre- vagy keresztél. Ezek mindegyikéről tudjuk (élek osztályozása, részfa-lemma utáni feladat), hogy w_j bejárása előbb fejeződik be, mint w_i bejárása, vagyis $j =$ bszám$[w_j] <$ bszám$[w_i] = i$. A $w_n, \ldots, w_1$ sorrendben így w_i megelőzi w_j-t. □

Legrövidebb és leghosszabb utak

Mint már említettük, a DAG-ok egyik előnyös tulajdonsága, hogy több fontos velük kapcsolatos feladatra gyorsabb algoritmusok ismeretesek, mint irányított gráfokra általában. Nézzük például a legrövidebb utak problémáját egy forrással. Legyen $G = (V, E)$ egy éllistával adott, súlyozott élű DAG és $s \in V$. Az élsúlyok tetszőleges valós számok lehetnek. G-ben nincs negatív kör, mivelhogy egyáltalán nem tartalmaz irányított kört. A feladatot a Bellman–Ford-módszerrel $O(n^3)$ lépésben tudjuk megoldani; nemnegatív élsúlyok esetén pedig a Dijkstra-algoritmus időigénye $O((n + e) \log n)$.

Topologikus rendezést használva a feladat lineáris időben, azaz $O(n + e)$ lépésben megoldható. Tegyük fel tehát, hogy már meghatároztuk a G csúcsainak egy $x_1, x_2, \ldots, x_n$ topologikus rendezését. Feltehetjük, hogy $s = x_1$, mert csak a kisebbtől a nagyobb sorszámú csúcsok felé mehet él. A $d(s, x_i)$ távolságokra érvényes a következő összefüggés:

$$(\dagger) \qquad d(s, x_i) = \min_{(x_j, x_i) \in E} \{d(s, x_j) + c(x_j, x_i)\},$$

ahol $c(x_j, x_i)$ az $x_j \to x_i$ él súlya. Ezt használva $i = 1, 2, \ldots n$-re sorban kiszámíthatjuk a $d(s, x_i)$ távolságokat. A lényeges mozzanat az, hogy ha (x_j, x_i) él, akkor $j < i$, tehát $d(s, x_i)$ számításakor a $d(s, x_j)$ értékek már tényleg a rendelkezésünkre állnak.

Mibe kerül mindez? Az i-edik távolság számításakor b_i számú összeadást végzünk el, ahol b_i az x_i-be befutó élek száma. A minimum meghatározására költött összehasonlítások száma is legfeljebb b_i. Ezeket i-re összegezve: e összeadásra és legfeljebb e összehasonlításra van szükség. Meg kell oldanunk még egy szervezési problémát: adott i-re gyorsan el kell tudnunk érni az x_i csúcsba bemenő éleket. Erről szól a következő feladat.

Feladat: Mutassuk meg, hogy G éllistájából $O(n + e)$ lépésben megkaphatjuk a *fordított éllistáját*. Utóbbiban az x_i csúcs listáján az x_i-be menő éleket leíró cellák vannak (szemben az eredetivel, ahol is x_i listája az x_i-ből kiinduló éleket ábrázolja).

Összesen tehát – a topologikus rendezést és a listafordítást is beleértve – $O(n + e)$ művelettel kiszámíthatjuk a legrövidebb utak hosszát.

A legrövidebb utak helyett érdekelhetnek bennünket a másik végletet jelentő *leghosszabb utak* is. Egy súlyozott élű G irányított gráf u, v pontjaira legyen $l(u, v)$ a leghosszabb *egyszerű* irányított $u \rightsquigarrow v$ utak hossza. Azért szorítkozunk egyszerű utakra, mert különben a pozitív összsúlyú irányított kört tartalmazó gráfokra a definíció értelmetlen volna.

A teendők elvégezhetőségének tehát tényleg szükséges és elegendő feltétele, hogy a hozzájuk rendelt gráf DAG legyen. Hasonló mondható a tranzakciós példánkról: a várakozási gráf topologikus rendezhetőségéből következik, hogy valamelyik T biztosan megkaphatja a hőn áhított zárat.

A tétel gondolatmenetében egy DAG-ok topologikus rendezésére szolgáló algoritmus lappang: válasszunk ki első pontnak egy olyan csúcsot, amibe nem fut él, majd a maradékból egy ilyen csúcsot másodikként; és így tovább. Az ötlet a hatékonyan kivitelezhető a *sor* adatszerkezet segítségével. A sor alapja egy *elem*ekből álló kettősen láncolt lista. Két alapművelet van: sorba(v, Q) a v elemet a Q sor végére illeszti; első(Q) pedig visszaadja és egyben kitörli Q-ból annak első elemét. Nyilvánvaló, hogy ezek a műveletek a lista elejét, illetve végét jelölő mutatókkal $O(1)$ költséggel megvalósíthatók. A sort szokás még FIFO-listának (first in, first out) is nevezni.

Most egy Q sort használó topologikus rendező algoritmus vázlata következik.

1. A (kezdetben üres) Q sorba illesszük be a G azon csúcsait, amelyekbe nem megy él.
2. Ha Q üres, akkor álljunk meg, különben legyen $u :=$ első(Q), és írjuk ki az u csúcsot.
3. Töröljük a gráfból az $u \to v$ éleket. Ha egy itt előforduló v csúcsba már nem megy él, akkor sorba(v, Q).
4. Menjünk vissza a 2. lépéshez.

Feladat: Mutassuk meg, hogy az előző algoritmus kiírási sorrendje tényleg a G topologikus rendezése, és hogy az eljárás $O(n + e)$ költséggel megvalósítható.

Ugyancsak lineáris költségű topologikus rendező eljárást kaphatunk a mélységi bejárás segítségével: végezzük el a G DAG egy mélységi bejárását, és írjuk ki G csúcsait a befejezési számaik szerint növekvő $w_1, \dots, w_n$ sorrendben. Ez megtehető az $O(n + e)$ költségkorlát megtartása mellett. A w csúcsot akkor írjuk ki, amikor a bejárása befejeződött, vagyis az mb(w) hívás utolsó utasításaként.

Állítás: *A $w_n, w_{n-1}, \dots, w_1$ sorrend a G DAG egy topologikus rendezése.*

Bizonyítás: Azt kell belátnunk, hogy ha $w_i \to w_j$ éle G-nek, akkor $i > j$. Ez azon múlik, hogy G-ben nincs visszaél. A $w_i \to w_j$ él tehát vagy fa- vagy előre- vagy keresztél. Ezek mindegyikéről tudjuk (élek osztályozása, részfa-lemma utáni feladat), hogy w_j bejárása előbb fejeződik be, mint w_i bejárása, vagyis $j =$ bszám$[w_j] <$ bszám$[w_i] = i$. A $w_n, \dots, w_1$ sorrendben így w_i megelőzi w_j-t. □

Legrövidebb és leghosszabb utak

Mint már említettük, a DAG-ok egyik előnyös tulajdonsága, hogy több fontos velük kapcsolatos feladatra gyorsabb algoritmusok ismeretesek, mint irányított gráfokra általában. Nézzük például a legrövidebb utak problémáját egy forrással. Legyen $G = (V, E)$ egy éllistával adott, súlyozott élű DAG és $s \in V$. Az élsúlyok tetszőleges valós számok lehetnek. G-ben nincs negatív kör, mivelhogy egyáltalán nem tartalmaz irányított kört. A feladatot a Bellman–Ford-módszerrel $O(n^3)$ lépésben tudjuk megoldani; nemnegatív élsúlyok esetén pedig a Dijkstra-algoritmus időigénye $O((n + e) \log n)$.

Topologikus rendezést használva a feladat lineáris időben, azaz $O(n + e)$ lépésben megoldható. Tegyük fel tehát, hogy már meghatároztuk a G csúcsainak egy $x_1, x_2, \ldots, x_n$ topologikus rendezését. Feltehetjük, hogy $s = x_1$, mert csak a kisebbtől a nagyobb sorszámú csúcsok felé mehet él. A $d(s, x_i)$ távolságokra érvényes a következő összefüggés:

$$(\dagger) \qquad d(s, x_i) = \min_{(x_j, x_i) \in E} \{d(s, x_j) + c(x_j, x_i)\},$$

ahol $c(x_j, x_i)$ az $x_j \to x_i$ él súlya. Ezt használva $i = 1, 2, \ldots n$-re sorban kiszámíthatjuk a $d(s, x_i)$ távolságokat. A lényeges mozzanat az, hogy ha (x_j, x_i) él, akkor $j < i$, tehát $d(s, x_i)$ számításakor a $d(s, x_j)$ értékek már tényleg a rendelkezésünkre állnak.

Mibe kerül mindez? Az i-edik távolság számításakor b_i számú összeadást végzünk el, ahol b_i az x_i-be befutó élek száma. A minimum meghatározására költött összehasonlítások száma is legfeljebb b_i. Ezeket i-re összegezve: e összeadásra és legfeljebb e összehasonlításra van szükség. Meg kell oldanunk még egy szervezési problémát: adott i-re gyorsan el kell tudnunk érni az x_i csúcsba bemenő éleket. Erről szól a következő feladat.

Feladat: Mutassuk meg, hogy G éllistájából $O(n + e)$ lépésben megkaphatjuk a *fordított éllistáját*. Utóbbiban az x_i csúcs listáján az x_i-be menő éleket leíró cellák vannak (szemben az eredetivel, ahol is x_i listája az x_i-ből kiinduló éleket ábrázolja).

Összesen tehát – a topologikus rendezést és a listafordítást is beleértve – $O(n + e)$ művelettel kiszámíthatjuk a legrövidebb utak hosszát.

A legrövidebb utak helyett érdekelhetnek bennünket a másik végletet jelentő *leghosszabb utak* is. Egy súlyozott élű G irányított gráf u, v pontjaira legyen $l(u, v)$ a leghosszabb *egyszerű* irányított $u \rightsquigarrow v$ utak hossza. Azért szorítkozunk egyszerű utakra, mert különben a pozitív összsúlyú irányított kört tartalmazó gráfokra a definíció értelmetlen volna.

Az $l(u, v)$ mennyiségek meghatározása általában hírhedten időigényes feladat, aminek néhány rokonával (Hamilton-kör keresése, Utazó ügynök) találkozni fogunk a 8.7.5. részben. A nehéz feladat megszelídül, ha a bemenetet jelentő gráf DAG. Először is megjegyezzük, hogy ha G DAG, akkor az $l(u, v)$ definíciójában *egyszerű út* helyett elég szimplán csak *utat* írni. Egy DAG-ban ugyanis minden irányított út egyszerű. A leghosszabb utakra ezután nyilvánvalóan érvényes a (†) rekurzió megfelelője – minimumok helyett maximumokkal:

$$(\ddagger) \qquad l(s, x_i) = \max_{(x_j, x_i) \in E} \{l(s, x_j) + c(x_j, x_i)\}.$$

Ha G egy DAG, akkor a (‡) összefüggések alapján az $l(s, x_i)$ mennyiségeket lényegében ugyanazzal a módszerrel megkaphatjuk, mint a $d(s, x_i)$ távolságokat. A különbség csupán annyi, hogy minimumok helyett maximumokat kell számítani.

Tétel: *Ha G egy éllistával adott súlyozott élű DAG, akkor az egy forrásból induló legrövidebb és leghosszabb utak meghatározásának feladatai $O(n + e)$ lépésben megoldhatók.* □

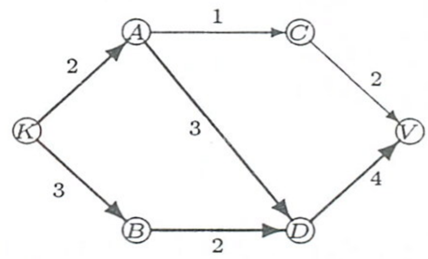
Feladat: Adjunk $O(n + e)$ lépésszámú módszert a legrövidebb és a leghosszabb utak nyomonkövetésére. (Használjuk a Dijkstra-módszernél alkalmazott gondolatot.)

A PERT-módszer

A PERT (Program Evaluation and Review Technique) elnevezésű tervezési módszertan súlyozott élű DAG-okkal ábrázolja egy összetett feladat részei közötti viszonyokat. A DAG csúcsai a nagy feladat részei, eleminek mondható teendői. Az x teendőből irányított él vezet az y teendőbe, ha x-et előbb kell elvégezni, mint y-t. A PERT-gráfokban külön csúcsok – legyenek ezek mondjuk K és V – jelölik a teljes feladat kezdetét és végét. Ha a gráf éleit értelmesen definiáljuk, akkor a csúcsok minden topologikus rendezésekor K az első, V pedig az utolsó csúcs lesz. Az éleken nemnegatív súlyok vannak. Az $x \to y$ él $c(x, y)$ súlya mondja meg, hogy mennyi időnek kell eltelnie az x tevékenység megkezdésétől az y megkezdéséig. Ilyen természetű Krúdy receptjében az a kikötés, hogy a salétromozás és a pácolás között egy napnak el kell telnie.

Hogyan határozhatjuk meg a súlyozott gráf ismeretében az x részfeladat legkorábbi kezdési időpontját? A válasz egyszerű: ez a leghosszabb $K \rightsquigarrow x$ út hossza, azaz $l(K, x)$ lesz. Speciális esetként a teljes feladat elvégzésének minimális időigénye $l(K, V)$. Ezek az idők az előző szakasz algoritmusával lineáris időben, vagyis $O(n + e)$ lépésben megkaphatók.

A PERT-módszertan fontos fogalmai a *kritikus út, él* és *csúcs*. Kritikus úton olyan $K \rightsquigarrow V$ utat értünk, amelynek a hossza $l(K, V)$. A kritikus élek és csúcsok pedig a kritikus utakon levő élek, illetve csúcsok. Az elnevezés onnan ered, hogy ha a kritikus utakon késve kezdődnek a tevékenységek, ez az egész munka befejezését késlelteti. A következő rajzon egy PERT-gráf szerepel. A kritikus éleket megvastagítottuk.



A C részfeladat nem kritikus. A legkorábbi kezdési ideje $l(K, C) = 3$, de az egész projekt befejezhető 9 időegység alatt akkor is, ha C-t a K után 7 egységgel indítjuk. A D csúcs viszont kritikus. Ha csak kicsivel is több, mint 5 időegység elteltével indítjuk, akkor az egész munka már nem fejezhető be 9 időegység alatt.

A kritikus csúcsokat és éleket a G PERT-gráf topologikus rendezésében visszafele, azaz V-től K felé haladva határozhatjuk meg. Első lépésként megjelöljük V-t mint kritikus csúcsot. Általában, tegyük fel, hogy az x_i csúcsnál tartunk, és x_i megjelölt (vagyis kritikus) csúcs. Ez esetben az $x_j \to x_i$ élet és az x_j csúcsot akkor kell megjelölnünk, ha (‡)-ban az $x_j \to x_i$ élen keresztül elérjük az $l(K, x_i)$ maximumot. Máshogy mondva: ha $l(K, x_i) = l(K, x_j) + c(x_j, x_i)$ teljesül. Az eljárás során minden élet egyszer veszünk szemügyre, így a költség $O(n + e)$. Érvényes tehát a következő:

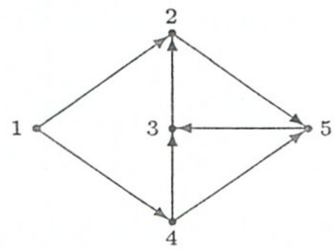
Tétel: *Egy éllistával adott PERT-gráf esetén a részfeladatok legkorábbi kezdési ideje, ezenfelül a kritikus élek és csúcsok lineáris időben meghatározhatók.* □

6.4.3. Erősen összefüggő (erős) komponensek

Irányított gráfoknál az összefüggőséget ugyanúgy definiáljuk, mint az irányítatlan gráfoknál: tekintet nélkül az élek irányítására. Ebben az értelemben az összefüggőségre úgy gondolhatunk, hogy a gráfunk egy darabból van. Irányított gráfokra van azonban egy ennél jóval szigorúbb összefüggőség-fogalom is:

Definíció (erősen összefüggő gráf): *Egy* $G = (V, E)$ *irányított gráf* erősen összefüggő, *ha bármely* $u, v \in V$ *pontpárra létezik* $u \rightsquigarrow v$ *irányított út.*

Ha valaki tévelygett már autóval egy idegen város egyirányú utcáinak szövevényében, akkor aligha fogja vitatni, hogy ez a követelmény erősebb, mint a „gyalogosokra szabott" egyszerű összefüggőség. A következő gráf összefüggő, de nem erősen összefüggő. Az 1 csúcsba nem juthatunk el irányított úton egyetlen más csúcsból sem.



Egy gráf összefüggő komponenseit a gráf csúcsain értelmezett ekvivalenciareláció ($u \sim v$, ha van u-ból v-be menő út) osztályaiként értelmeztük. Hasonlóan járhatunk el, ha az elérhetőségnél az élek irányát is figyelembe akarjuk venni.

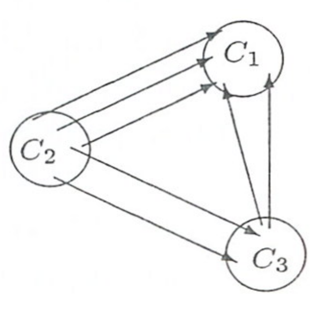
Definíció: *Legyen $G = (V, E)$ egy irányított gráf. Bevezetünk egy relációt V-n: $u, v \in V$-re legyen $u \approx v$, ha G-ben léteznek $u \rightsquigarrow v$ és $v \rightsquigarrow u$ irányított utak.*

Természetesen $v \approx v$, hiszen a nulla hosszúságú utat is útnak tekintjük. Nyilvánvaló az is, hogy a $\approx$ reláció szimmetrikus és tranzitív is. Következésképpen ekvivalenciareláció, ami osztályokba sorolja a gráf pontjait.

Definíció (erős komponensek): *A $\approx$ reláció ekvivalenciaosztályait a G* erősen összefüggő (*röviden:* erős) komponensei*nek nevezzük.*

A definícióból azonnal látható, hogy ha C a G-nek erős komponense, akkor a C a pontjait összekötő élekkel együtt egy erősen összefüggő gráf. Az előző példa gráfjának három erős komponense van: $\{1\}$, $\{4\}$ és $\{2, 3, 5\}$.

Állítás: *Egy irányított gráf két erős komponense között az élek csak egy irányba mehetnek.*



Bizonyítás: Ha menne él a C_1 erős komponensből egy másik C_2-be és a C_2-ből a C_1-be is, akkor C_1 és C_2 ugyanabban az erős komponensben volna. □

Így ha egy irányított gráfban csak az erős komponensek viszonyára vagyunk kíváncsiak, akkor elég nyilvántartanunk, hogy mely komponensek között vezet él, és milyen irányban.

Definíció (redukált gráf): *Legyen $G = (V, E)$ irányított gráf. G redukált gráfja egy irányított gráf, melynek pontjai a G erős komponensei; a C_1, C_2 komponensek között akkor van $C_1 \to C_2$ él, ha G-ben a C_1 komponens valamely pontjából vezet él a C_2 komponensbe.*

A redukált gráf mindig DAG lesz. Ugyanis egy $C_1 \to C_2 \to \cdots \to C_k \to C_1$ irányított kör a redukált gráfban azt jelentené, hogy $C_1 \cup C_2 \cup \cdots \cup C_k$ a G ugyanazon erős komponensében van.

Most egy gyors (lineáris idejű) módszert mutatunk egy éllistával adott $G = (V, E)$ irányított gráf erős komponenseinek a feltérképezésére. Az algoritmus a mélységi bejáráson alapul, amit mindjárt nem is egyszer, hanem kétszer hívunk segítségül.

Erősen összefüggő komponensek meghatározása

(1) Mélységi bejárással végigmegyünk G-n, közben minden pontnak sorszámot adunk: a befejezési számát.
(2) Elkészítjük a G_{ford} gráfot, melyet úgy kapunk G-ből, hogy minden él irányítását megfordítjuk. Pontosabban: $G_{\text{ford}} := (V, E')$, ahol $u \to v \in E'$ akkor és csak akkor, ha $v \to u \in E$.
(3) Bejárjuk a G_{ford} gráfot mélységi bejárással, a legnagyobb sorszámú csúccsal kezdve (az (1)-beli befejezési számozás szerint). Új gyökérpont választásakor mindig a legnagyobb sorszámú csúcsot vesszük a maradékból.

Tétel: *A (3) pontban kapott fák lesznek G erős komponensei, azaz G-ben $x \approx y$ pontosan akkor igaz, ha x és y egy fában vannak.*

Bizonyítás: $\Rightarrow$: Azt kell igazolnunk, hogy egy erős komponens pontjai egy fába kerülnek a (3)-beli bejárás során. Ennél valamivel több is igaz: a gráf egy erős komponensének a pontjai *bármely* mélységi bejárásnál egyazon fába kerülnek. Ezt a G_{ford} gráfra alkalmazva – aminek nyilván ugyanazok az erős komponensei, mint G-nek – adódik az állítás.

Legyen tehát K egy erős komponens, és legyen x a K legkisebb mélységi számú pontja. Ekkor nyilván $K \subseteq S_x$, így a részfa-lemma miatt K benne van az x-ben gyökerező részfában.

$\Leftarrow$: Fordítva, tegyük fel, hogy x és y egy fában vannak a (3) pont szerinti mélységi bejárás után. Azt kell belátnunk, hogy ekkor $x \approx y$ a G gráfban, azaz x és y egymásból irányított úton elérhetők.

Legyen a v csúcs a gyökere annak a fának, mely a (3) pont szerinti mélységi bejárás után x-et és y-t is tartalmazza. Mivel x leszármazottja v-nek, ezért a G_{ford} gráfban van $v \rightsquigarrow x$ irányított út, tehát a G gráfban van egy L irányított út x-ből v-be.

Legyen x' az L-nek az a pontja, amelynek az első bejárás szerinti mélységi száma a legkisebb. A részfa-lemma miatt L-nek az $x' \rightsquigarrow v$ darabjában levő csúcsok az (1) bejárásnál x' leszármazottai lesznek. Az x' gyökerű részfában viszont az x' befejezési száma a legnagyobb, így v választása miatt $x' = v$. Az L pontjai közül tehát v-t látogattuk meg legelőször, és v-nek a befejezési száma volt a legnagyobb. Ezek együtt azt jelentik, hogy L pontjai a v leszármazottai az első bejárásnál. Így G-ben van $v \rightsquigarrow x$ irányított út.

Beláttuk, hogy a G gráfban $x \approx v$. Hasonlóan adódik, hogy $y \approx v$. A reláció szimmetrikus és tranzitív volta miatt $x \approx y$, azaz x és y a G egyazon erős komponensében vannak. □

Az első lépés $O(n+e)$ időt vesz igénybe. A gráf bejárása alatt kiírjuk a csúcsait befejezési szám szerinti sorrendben. (Emiatt a (3) lépésnél az új gyökér választásához nem kell külön rendezni őket.) A második lépés $O(e)$, a harmadik pedig $O(n + e)$ ideig tart. Tehát az algoritmus összidőigénye $O(n + e)$.

Feladat: Írjuk fel a G erős komponenseit abban a sorrendben, ahogy az előző algoritmus adja őket. Mutassuk meg, hogy az így kapott sorrend a G redukált gráfjának egyik topologikus rendezése.

6.4.4. Irányítatlan gráfok mélységi bejárása

A mélységi bejárás irányítatlan gráfokkal kapcsolatos feladatok megoldásában is hasznos. Egy irányítatlan gráf felfogható irányítottként úgy, hogy minden (u, v) élét az $u \rightarrow v$ és $v \rightarrow u$ élekkel helyettesítjük. Ezzel az átalakítással a 6.4.1. részben elmondottak irányítatlan gráfokra is értelmezhetők.

Az irányított gráfoknál bemutatott algoritmus tehát minden változtatás nélkül itt is működik. Segítségével a csúcsok számozásai (mélységi és befejezési), az élek osztályozása, a mélységi feszítő erdő $O(n + e)$ költséggel megkaphatók. A következő két észrevétel azt mondja, hogy irányítatlan gráfoknál a bejárással kapcsolatos tulajdonságok egyszerűbbek:

– Amikor a G irányítatlan gráfot irányítottá alakítjuk, az összefüggő komponenseiből erősen összefüggő gráfok lesznek. Ezért ha a C komponensből való x csúcsot választjuk a bejárás kezdőpontjául, akkor a részfa-lemma Következménye szerint az x részfájának csúcshalmaza pontosan C lesz. A mélységi feszítő erdő fái így a G összefüggő komponenseinek felelnek meg. A mélységi bejárás tehát irányítatlan gráfok komponenseinek feltérképezésére is alkalmas. Speciális esetként: ha G összefüggő, akkor $O(e)$ költséggel egy feszítőfát kapunk.

– Irányított gráfok bejárásakor négyféle éllel találkoztunk: fa-, előre-, vissza- és keresztéllel. Irányítatlan gráfoknál ebben is egyszerűbb a kép. A (v, w) irányítatlan él típusa legyen a $v \to w$ és $w \to v$ irányított élek közül annak a típusa, *amelyiket a bejáráskor először vizsgáljuk meg*. Az élek típusát így értelmezve egy irányítatlan gráf mélységi bejárásánál *csak faélek és visszaélek lehetnek*. Ennek bizonyítására vegyünk egy tetszőleges (v, w) élet. Az általánosság csorbítása nélkül feltehetjük, hogy az mb() eljárást a v csúcsra hívjuk meg előbb. Az mb(v) hívás során biztosan megvizsgáljuk a $v \to w$ élet. Ha a w-t eme vizsgálat előtt még nem látogattuk meg, akkor $v \to w$ és ezért (v, w) is faél lesz. Ellenkező esetben w az mb(v) eljárás folyamán vált látogatottá, tehát w leszármazottja lesz v-nek a fában. Ez esetben a w bejárása fejeződik be előbb; a $w \to v$ élet előbb látjuk, mint a fordított párját. Ennélfogva (v, w) visszaél lesz.

Alkalmazás: Artikulációs pont keresése

Definíció: *Legyen $G = (V, E)$ összefüggő irányítatlan gráf. A $v \in V$ csúcs* artikulációs *(más szóval* elvágó*) pontja G-nek, ha v és a rá illeszkedő élek elhagyásával a gráf több komponensre esik szét, vagyis elveszti összefüggőségét.*

Egy gráf artikulációs pontjait a mélységi bejárás segítségével találhatjuk meg. Hogy megértsük a következő módszert, képzeljünk magunk elé egy összefüggő gráfot, melynek már felépítettük a mélységi feszítőfáját. A gráf tehát csak faéleket és visszaéleket tartalmaz. Az utóbbiak a fa egy ágának két csúcsát kötik össze. Mivel keresztélek nincsenek, a fa két különböző ágát nem köti össze él. Ebből rögtön következik, hogy a fa gyökere pontosan akkor artikulációs pontja a gráfnak, ha egynél több fia van, hiszen ekkor elhagyásával a gráf a gyökér alatti részfáknak megfelelő darabokra esik szét.

Most vizsgáljuk meg, hogy mi történik, ha elhagyunk egy gyökértől különböző v csúcsot a gráfból. Ha csak a faéleket nézzük, akkor azt látjuk, hogy a feszítőfa szétesik azon részfákra, melyeknek gyökerei a v fiai, és a maradék feszítőfadarabra (ilyen van, mert v nem a gyökér). A visszaélek csak úgy tarthatják egybe ezeket a darabokat, ha a v alatti nem üres részfák mindegyikéből megy visszaél a v feletti feszítőfadarabba. Ez azt jelenti, hogy v-nek bármelyik ágán is indulunk el „lefelé"

a feszítőfában, egy visszaélen mindig „fel" tudunk jutni v egy valódi ősébe. Ha v-nek van akár egyetlen olyan fia, melynek részfájából nem vezet visszaél v „fölé", akkor v elhagyásával ez a részfa biztosan le fog válni a gráfról.

A $v \in V$ csúcsról el akarjuk dönteni, hogy vajon minden részfájából vezet-e visszaél egy valódi ősébe. E célból kiszámítjuk a fel$[v]$ értéket. Ez megadja a v csúcshoz annak a „feszítőfában legmagasabban levő" w csúcsnak a mélységi számát, amelyhez el tudunk jutni v-ből úgy, hogy „lefelé" megyünk (esetleg 0 darab) faélen, aztán egy visszaélen „felmegyünk" w-be. Legrosszabb esetben ez a csúcs maga v, különben pedig w mélységi száma kisebb lesz v mélységi számánál, hiszen ekkor w valódi őse v-nek. A v csúcs tehát akkor és csak akkor lesz artikulációs pont, ha van olyan w fia, melynek a fájából nem megy v fölé visszaél, vagyis melyre fel$[w] \geq$ mszám$[v]$. Ezek után foglaljuk össze egy kicsit gondosabban a tennivalókat.

Módszer az artikulációs pontok megkeresésére (összefüggő gráfban):

1. Végezzük el a gráf mélységi bejárását, és határozzuk meg a csúcsok mélységi számát, melyet $v \in V$-re mszám$[v]$ jelöl.

2. Számítsuk ki minden v csúcsra a fel$[v]$ értéket; ez megadja annak a legkisebb mélységi számú w csúcsnak a mélységi számát, melybe vezet visszaél v valamely leszármazottjából. Hogy ezt megvalósítsuk, járjuk be az első pontban kapott feszítőfát a befejezési számok szerinti sorrendben, és ebben a sorrendben töltsük ki a fel[] tömböt. Így amikor a fel$[v]$ értéket próbáljuk meghatározni, akkor v minden y fiára fel$[y]$ már ismert.

$$\text{fel}[v] = \min \left\{ \begin{array}{l} \text{mszám}[v], \\ \min\{\text{mszám}[z], \text{ ahol } v \to z \text{ visszaél}\}, \\ \min\{\text{fel}[y], \text{ ahol } y \text{ fia } v\text{-nek}\} \end{array} \right\}$$

3. Artikulációs pontok megkeresése: a feszítőfát bejárva a $v \in V$ csúcsokról ellenőrizzük, hogy elvágó pontok-e. A v-re vonatkozó teszt így hangzik:

(a) a gyökér pontosan akkor artikulációs pont, ha legalább 2 fia van a fában.

(b) a gyökértől különböző v csúcs akkor és csak akkor artikulációs pont, ha van v-nek olyan y fia, hogy fel$[y] \geq$ mszám$[v]$.

Mindhárom lépés gyakorlatilag gráf-, illetve fabejárás, ami $O(n + e)$ időt vesz igénybe. A gráf összefüggő, tehát $e \geq n - 1$, amiből az algoritmus összköltsége $O(e)$.

6.5. A szélességi bejárás

Elevenítsük föl a 6.4.-beli esetet, amikor a lámpagyújtogató elhívta a barátait lámpát gyújtogatni. Az általuk követett bejáró módszert próbáljuk meg átültetni tet-

szőleges irányított gráfra. Olyan bejárási sorrendet akarunk, ami szerint csak akkor foglalkozunk a kezdőponttól távolabbi csúcsokkal, ha a közelieket már mind láttuk. Meglátogatjuk tehát az első csúcsot, majd ennek a csúcsnak az összes szomszédját. Aztán ezen szomszédok összes olyan szomszédját, hol még nem jártunk, és így tovább. Hogyan lehet ezt értelmesen megszervezni? A legjobb ismert megoldás egy *sort* (FIFO-listát) alkalmaz. Ebbe rakjuk be az épp meglátogatott csúcsot, hogy majd a megfelelő időben a szomszédaira is sort keríthessünk.

A módszer általános lépésének lényege, hogy vesszük a sor elején levő x csúcsot. Ezt töröljük a sorból, meglátogatjuk azokat az y szomszédait, amelyeket eddig még nem láttunk, majd ezeket az y csúcsokat a sor végére tesszük. Az algoritmus keretét adó *bejár* eljárás tulajdonképpen ugyanaz, mint a mélységi bejárásé. A bejárva$[1 : n]$ tömb szolgál itt is a már látott csúcsok megjelölésére.

```
procedure bejár (* elvégzi a G irányított gráf szélességi bejárását *)
    begin
        for v := 1 to n do
            bejárva[v] := hamis;
        for v := 1 to n do
            if bejárva[v] = hamis then
              szb(v)
    end

procedure szb (v: csúcs)
    var
        Q: csúcsokból álló sor;
        x, y: csúcsok;
    begin
        bejárva[v] := igaz;
        sorba(v, Q);
        while Q nem üres do begin
            x := első(Q);
            for minden x → y ∈ E élre do
                if bejárva[y] = hamis then begin
                    bejárva[y] := igaz;
(*)                 sorba(y, Q)
                end
        end
    end
```

A szélességi bejárás során – a mélységi mintájára – felépíthetjük a gráf egy *szélességi feszítő erdejét*. *Faélnek* itt is azokat az éleket hívjuk, melyek megvizsgálásukkor még be nem járt pontba mutatnak. Tehát amikor egy már meglátogatott x csúcs szomszédait vizsgáljuk, és mondjuk y-t közülük még nem láttuk, akkor az egyre növekvő erdőhöz hozzávesszük az $x \to y$ élet (az algoritmusban a $(*)$ sor után). Csakúgy, mint a mélységi bejárásnál, a faélek itt is feszítő erdőt alkotnak, aminek a fái természetes módon tekinthetők gyökeres, a gyökértől távolodóan irányított fáknak. E fák kapcsán most is osztályozhatjuk a gráf éleit.

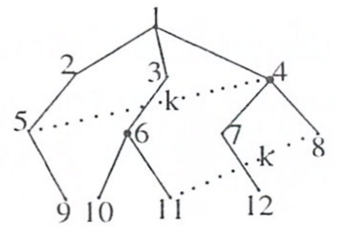
A G irányított gráf szélességi bejárása során a gráf egyik éle sem bizonyulhat előreélnek. Ugyanis amikor eljárásunk az x csúcsot kiveszi a sorból, hogy szomszédait megvizsgálja, akkor x minden még bejáratlan y szomszédját bejárttá nyilvánítja, a sor végére teszi, és az $x \to y$ él így faél lesz. A már bejárt y szomszédok pedig vagy x ősei, vagy egy másik ágon (esetleg egy másik fában) találhatók. Az előbbi esetben az $x \to y$ él *visszaél*, az utóbbiban pedig *keresztél* lesz.

A szélességi bejárás alkalmazható irányítatlan gráfokra is. A szokásos módon az (x, y) irányítatlan élet az $x \to y$ és az $y \to x$ irányított élpár helyettesíti. Az élek osztályozása a mélységi bejárásnál látottakhoz hasonlóan végezhető: az (x, y) irányítatlan él típusa legyen az $x \to y$ és $y \to x$ irányított élek közül annak a típusa, amelyikkel a bejáráskor először találkozunk. Ezzel a megállapodással élve az irányítatlan gráfok szélességi bejárása során *faéleken kívül csak keresztélek keletkeznek*. Nem szerepelhet előre- vagy visszaél. Legyen ugyanis (x, y) éle G-nek, és tegyük fel, hogy x őse y-nak (valamelyik fában). Nézzük, mi a helyzet akkor, amikor x kikerül a Q sorból. Ekkor y-ban még nem járhattunk, hiszen x fiai és így az összes leszármazottai csak ezután következnek. Tehát ekkor bejárva$[y] =$ $hamis$; emiatt az algoritmus (x, y)-t faélnek választja.

Feladat: Legyen G irányítatlan gráf. Mutassuk meg, hogy G bármely szélességi feszítő erdejében a fák ponthalmazai éppen a G összefüggő komponensei. (Legyen v az egyik fa gyökere, w pedig tetszőleges pont a G-nek a v-t tartalmazó komponenséből. Tekintsünk egy v-ből w-be menő utat, és mutassuk meg, hogy ennek a pontjai a szélességi bejárás alatt mind bekerülnek a v fájába.)

A szélességi bejárás költsége $O(n+e)$, mert minden csúcsot pontosan egyszer teszünk be a sorba (amikor látogatottra állítjuk), és minden irányított élet egyszer vizsgálunk meg (amikor a kezdőpontja kikerül a sorból).

A következő ábrán egy összefüggő irányítatlan gráf szélességi feszítőfája látható. A faéleket sima, a keresztéleket szaggatott vonallal jelöltük. A csúcsok melletti szám pedig azt mondja meg, hogy hányadikként látogattuk meg őket (*szélességi számozás*).



Még egyszer a legrövidebb utakról

A szélességi bejárás alkalmazásával az egy forrásból induló legrövidebb utak feladata (lásd 6.2.) lineáris időben megoldható, ha az élsúlyok egységnyiek.

Legyen $G = (V, E)$ egy irányított gráf, és $s \in V$ egy rögzített pontja. Az egyszerűség kedvéért tegyük fel, hogy s-ből minden $v \in V$ csúcs elérhető irányított úton. Megmutatjuk, hogy a szélességi bejárás segítségével a $d(s, v)$ távolságok ($v \in V$) lineáris időben meghatározhatók. Jelentse $D[v]$ a v csúcsnak az s-től való távolságát az s-gyökerű szélességi fában. A $D[v]$ mennyiségek a bejárás során a futási idő nagyságrendjének megváltoztatása nélkül kiszámíthatók. Legyen ugyanis kezdetben $D[s] := 0$; az *szb* eljárásba pedig a $(*)$ sor után tegyük be a kézenfekvő $D[y] := D[x] + 1$; utasítást. A kiegészített algoritmus futási ideje nyilvánvalóan $O(n + e)$.

Tétel: *Az előzőek szerint módosított szélességi bejárás végeztével teljesülnek a következők:*
1. Legyen $s = x_1, x_2, \ldots, x_n$ a csúcsoknak a szélességi bejárás szerinti sorrendje. Ekkor $D[x_1] \le D[x_2] \le \ldots \le D[x_n]$.
2. Ha $x \to y$ éle G-nek, akkor $D[y] \le D[x] + 1$.
3. $D[v] = d(s, v)$ teljesül minden $v \in V$ csúcsra.

Bizonyítás: 1. Az algoritmusra pillantva azonnal látszik, hogy a csúcsok az $s = x_1, x_2, \ldots, x_n$ sorrendben kerülnek bele a Q sorba. Következésképpen ebben a sorrendben is kerülnek ki a sorból (FIFO-tulajdonság). Innen arra jutunk, hogy ha az $x \neq s$ csúcs előbb van a sorrendben, mint y, akkor $apa(x)$ nem lehet később, mint $apa(y)$, ahol az apa a szélességi feszítőfában értendő. Ezt használva egyszerű indukcióval kapjuk, hogy a $D[x_1], D[x_2], \ldots, D[x_n]$ számsorozat nem csökkenő. Ez ugyanis közvetlenül látszik a gyökérre és fiaira. Később pedig nyilván $D[x_i] = D[apa(x_i)] + 1$ és $D[x_{i+1}] = D[apa(x_{i+1})] + 1$. Ha itt a két apa különböző, akkor az indukciós feltevés miatt $D[apa(x_i)] \le D[apa(x_{i+1})]$, amiből $D[x_i] \le D[x_{i+1}]$. Ha pedig az apák megegyeznek, akkor $D[x_i] = D[x_{i+1}]$.

2. Tegyük fel, hogy $x \to y$ éle G-nek; meg kell mutatnunk, hogy ekkor $D[y] \le D[x] + 1$. Nézzük ezért, hogy mi történik, amikor x kikerül a Q sorból, és éppen az (x, y) élet vizsgáljuk. Ha bejárva$[y]$ = *hamis*, akkor y apja x,

vagyis $D[y] = D[x] + 1$. Ha pedig y-t már korábban láttuk, akkor arra következtethetünk, hogy y apja előbb van a szélességi sorrendben, mint x, amiből 1. szerint $D[apa(y)] \leq D[x]$. Az utóbbi egyenlőtlenség mindkét oldalához egyet adva kapjuk, hogy $D[y] \leq D[x] + 1$.

3. Világos egyrészt, hogy $d(s, v) \leq D[v]$ érvényes minden $v \in V$ csúcsra, mivelhogy a gráfban (sőt a fában) van $D[v]$ élből álló $s \rightsquigarrow v$ irányított út. Elég tehát a fordított $D[v] \leq d(s, v)$ egyenlőtlenségeket igazolni. Legyen evégből $s = y_0, y_1, \ldots y_k = v$ egy minimális hosszúságú G-beli irányított út s-ből v-be. A 2. észrevételt alkalmazzuk sorra az út éleire: $D[y_1] \leq D[s] + 1 = 1$, majd $D[y_2] \leq D[y_1] + 1 \leq 2$; így folytatva végül $D[v] = D[y_k] \leq k = d(s, v)$. Az érvelés ezzel teljes. □

A $D[v]$ számok 3. szerint a legrövidebb s-ből v-be menő utak hosszai. A szélességi bejárással tehát a legrövidebb utak feladatának ez a fontos speciális esete (minden élsúly 1) a Dijkstra-algoritmusnál gyorsabban[3], lineáris időben megoldható.

A tétel 1. és 3. állításai mondják ki, hogy az eljárás tényleg a lámpagyújtogató és barátai széltében terjeszkedő stratégiáját valósítja meg: a kezdőpont után először a tőle 1 távolságra levő csúcsokat látogatjuk meg, utána a 2 egységnyire levők jönnek, és így tovább.

Feladat: Legyen G egy éllistával adott irányítatlan gráf, és s egy csúcsa. Adjunk $O(e)$ költségű módszert legrövidebb (legkevesebb élből álló) s-et tartalmazó kör keresésére. (Indítsunk szélességi bejárást s-ből. Legyenek $s_1, s_2, \ldots, s_k$ az s szomszédai. Legyen f a bejárás során kapott első olyan keresztél, amely az s_i gyökerű részfák közül két különböző között fut. Ekkor f a végpontjait összekötő faélekkel együtt megfelelő kört ad. Az f megtalálásában segít, ha a bejárás során a csúcsok mellé feljegyezzük, hogy melyik s_i részfájába tartoznak. Ezt a jellemzőt a csúcsok az apjuktól öröklik.)

A szélességi bejárás két további alkalmazásával később, a párosítások, majd pedig a hálózati folyamok kapcsán ismerkedünk meg.

[3]Talán helyesebb úgy fogalmazni, hogy a szélességi bejáráson alapuló módszerünk gyorsabb, mint a Dijkstra korábban megismert éllistás implementációja. Valójában a most bemutatott algoritmust szemlélhetjük úgy, mint Dijkstra-módszer egy változatát. A már bejárt pontok halmaza a KÉSZ halmaz, amit a faélek mentén bővítgetünk. A tétel 1. és 3. állításából kiolvasható, hogy teljesül a Dijkstra-módszer bővítési elve: a KÉSZ-be mindig egy az s-hez legközelebbi $V \setminus$KÉSZ-beli pont kerül.

6.6. Minimális költségű feszítőfák

egy fa ága vége
nem nő rá a gyökerére
se más ágra
se törzsére
(körmentes)

egy hangya ha elindulna
bármely ágról eljuthatna
bármely ágra
bármely pontba
(összefüggő)

hangyánk ha nem boldogulna
lepottyanva
földön mászna
törzsön kúszna
(ez bizony már erdő volna)

LEHEL JENŐ

A következő két részben irányítatlan gráfokkal (röviden: gráfokkal) foglalkozunk. Ennek megfelelően $G = (V, E)$ egy irányítatlan gráfot jelöl, amelynek n csúcsa és e éle van. Ezután már csak egyszerű utakkal és körökkel lesz dolgunk. A továbbiakban ezért úton és körön mindig *egyszerű* utat és kört értünk. Ebben a részben a minimális költségű feszítőfák keresésének problémáját fogjuk tárgyalni. Először tisztázzuk az alapfogalmakat.

Definíció (minimális költségű feszítőfa): *Legyen $G = (V, E)$ egy összefüggő gráf. A G gráf egy körmentes összefüggő $F = (V, E')$ részgráfja a gráf egy* feszítőfája. *Legyen továbbá az éleken értelmezve egy $c : E \to \mathbb{R}$ súlyfüggvény. Ekkor a G gráf egy F feszítőfája* minimális költségű, *ha költsége (a benne szereplő élek súlyainak összege) minimális G összes feszítőfája közül.*

Egy feszítőfa tehát egy olyan fa, ami a G minden pontját tartalmazza, és az élei a G élei közül valók.

A probléma

Adott egy $G = (V, E)$ összefüggő irányítatlan gráf, és az élein értelmezett $c : E \to \mathbb{R}$ súlyfüggvény. Határozzuk meg a G egy minimális költségű feszítőfáját.

A feladat természetesen merül fel olyan helyzetekben, amikor a G egy (u, v) éle az u és v közötti közvetlen kapcsolat lehetőségét jelenti, az él súlya pedig a kapcsolat kiépítésének, esetleg működtetésének valamiféle költségét. Egy olyan élrendszert keresünk, ami biztosítja, hogy bárhonnan eljuthatunk bárhová; ezek közül pedig a lehető legkisebb költségűt szeretnénk megtalálni (ld. a következő állítást is). Tudomásunk szerint a probléma első érdemi megoldását Otakar Borůvka brnói professzor közölte 1926-ban. A kérdéssel Morvaország nyugati részének a villamosítása kapcsán taláikozott: adott helyeket összekötő minimális összhosszúságú vezetékrendszert kellett terveznie.

Mielőtt az algoritmusok taglalásába fognánk, összefoglalunk néhány fákkal kapcsolatos egyszerű tényt.

Állítás:
1. Minden legalább kétpontú fában van olyan csúcs, amiből csak egy él megy ki (elsőfokú csúcs).
2. Bármely összefüggő gráf tartalmaz feszítőfát.
3. Egy n-pontú összefüggő gráf akkor és csak akkor fa, ha $n-1$ éle van.
4. Egy fa bármely két pontja között pontosan egy út vezet.
5. Legyen G egy súlyozott élű összefüggő gráf, F egy minimális költségű feszítőfája. Legyen $g=(u,v)$ a G-nek egy olyan éle, ami nem éle F-nek, és tegyük fel, hogy az F-beli u-ból v-be vezető úton van olyan g' él, amelyre $c(g) \leq c(g')$. Ekkor az F-ből a g hozzávételével és a g' elhagyásával adódó F' gráf is egy minimális költségű feszítőfa G-ben.

Bizonyítás: 1. A DAG-ok topologikus rendezhetőségéről szóló tétel ötlete működik itt is. Lépkedjünk a fa élei mentén, ügyelve arra, hogy ne használjuk kétszer egymás után ugyanazt az élet. A körmentesség miatt nem érhetünk vissza korábban már látott pontba. Így lesz olyan csúcs, amiből nem tudunk továbblépni. Ez egy elsőfokú csúcs.

2. Ha a gráfban van legalább 3 pontú egyszerű kör, akkor hagyjuk el ennek egy (u,v) élét. A gráf ezután is összefüggő marad; u-ból v továbbra is elérhető a körbeli kerülő úton. Ezt ismételgetve végül körmentes összefüggő gráfot, azaz egy feszítőfát kapunk.

3. Először belátjuk, hogy egy n-pontú fának $n-1$ éle van. Ugyanis ha a fából törlünk egy elsőfokú csúcsot a hozzá csatlakozó éllel együtt, akkor egy $n-1$-pontú fát kapunk. Ebből is törölhetünk egy ilyen lógó élet. Ezt ismételve $n-1$ lépés után az élek elfogynak, hiszen csak egy csúcs marad.

Fordítva: legyen F egy n-pontú, $n-1$-élű összefüggő gráf. Legyen F' ennek egy feszítőfája (lásd 2.). Az előzőleg igazolt állítás szerint F'-nek is $n-1$ éle van, amiből $F=F'$.

4. Tegyük fel, hogy az u pontból az u' pontba két út vezet. Legyen $f=(v,w)$ az egyik út olyan éle, ami nincs a másik úton. Nyilvánvaló ekkor, hogy f nélkül is eljuthatunk v-ből w-be, vagyis a gráf az f törlése után is összefüggő marad. Ez pedig 3. szerint képtelenség; egy fa az egyik élének a törlése után nem maradhat összefüggő.

5. Az $F \cup \{g\}$ gráfban van olyan kör, amelynek g' éle. A g' törlésével kapott F' gráf tehát összefüggő marad. A csúcsainak száma ugyanannyi, mint F-é, így 3. szerint egy feszítőfája G-nek. Végezetül pedig nyugtázzuk, hogy F' költsége nem lehet nagyobb, mint az F költsége, hiszen az előbbi az utóbbiból a nem pozitív $c(g)-c(g')$ mennyiség hozzáadásával kapható meg. □

A piros-kék algoritmus

A minimális feszítőfák keresésére való algoritmusok legtöbbje egy tőről származtatható. E módszerek közös vonása, hogy valamilyen módon sorra nézik a G éleit; bizonyosakat bevesznek a végül kialakuló minimális feszítőfába, másokat pedig eldobnak. A módszerek működését érdemes úgy szemlélni, hogy színezzük a G éleit[4]. Két színt használunk: a *kék* élek belekerülnek a végeredményt jelentő minimális feszítőfába, a *pirosak* pedig nem. Az élek festegetése során ügyelni fogunk arra, hogy az eddig kialakult (részleges) színezés mindig *takaros* legyen.

Definíció (takaros színezés): *Tekintsük a súlyozott élű G gráf éleinek egy részleges színezését, melynél bármely él piros, kék vagy színtelen lehet. Ez a színezés* takaros, *ha van G-nek olyan minimális költségű feszítőfája, ami az összes kék élet tartalmazza, és egyetlen piros élet sem tartalmaz.*

A színezés során két szabályt használunk. Ez annyit tesz, hogy egy élet csak akkor festünk be, ha a szabályok egyike alkalmazható.

KÉK SZABÁLY:	Válasszunk ki egy olyan $\emptyset \neq X \subset V$ csúcshalmazt, amiből nem vezet ki kék él. Ezután egy legkisebb súlyú X-ből kimenő színezetlen élet fessünk kékre.
PIROS SZABÁLY:	Válasszunk G-ben egy olyan egyszerű kört, amiben nincs piros él. A kör egyik legnagyobb súlyú színtelen élét fessük pirosra.

Kezdetben G-nek nincs színes éle. Ebből a helyzetből indulva a két szabályt tetszőleges sorrendben és helyeken alkalmazzuk, amíg csak lehetséges. Nevezzük ezt a nagyvonalú algoritmust *piros-kék algoritmus*nak. Első látásra talán meglepő, hogy mindenképpen célt érünk, függetlenül attól, hogy milyen sorrendben színezzük az éleket. Ezt fejezi ki a következő két állítás:

Tétel: *A piros-kék eljárás működése során mindig takaros színezésünk van. Ezen felül a színezéssel sosem akadunk el: végül G minden éle színes lesz.*

Bizonyítás: Először belátjuk, hogy a színezés mindig takaros. Ez kezdetben, amikor még minden él színtelen, nyilván teljesül. Tegyük fel mármost, hogy egy takaros színezésünk van. Legyen F a G egy olyan minimális költségű feszítőfája, amely minden kék élet tartalmaz, és egyetlen pirosat sem. Tegyük fel továbbá,

[4]Az algoritmusoknak ezt az interpretációját Robert E. Tarjan javasolta.

hogy ebben a helyzetben a gráf f élét festjük be. Két eset van aszerint, hogy melyik szabályt használjuk:
(a) *A kék szabályt használjuk.* Ekkor f nyilván kék lesz. Ha f éle F-nek, akkor F maga mutatja, hogy a színezés takaros marad. Ha f nem éle F-nek, akkor nézzük azt az $X \subset V$ csúcshalmazt, amire a kék szabályt alkalmaztuk. Az F-ben van olyan út (mert feszítőfa), ami az f két végpontját összeköti. Ezen az úton pedig van olyan f' él, ami kimegy X-ből, ugyanis f kilép X-ből. Az F választása miatt f' nem lehet piros. A kék szabály szerint kék sem lehet, továbbá $c(f') \geq c(f)$ is teljesül. Legyen F' az F-ből az f' törlésével és az f hozzáadásával kapott gráf. A $g = f$ és $g' = f'$ választással alkalmazható az állítás 5. pontja. Eszerint F' egy minimális feszítőfa. Az F' mutatja, hogy az f befestése utáni színezés is takaros.
(b) *A piros szabályt használjuk.* Ekkor f piros lesz. Ha f nem éle F-nek, akkor F tanúsítja a bővebb színezés takarosságát is. Ha $f \in F$, akkor az f törlésével az F két komponensre esik. Pillantsunk most arra a körre, amelyre a piros szabályt alkalmaztuk. Ennek van olyan $f' \neq f$ éle, ami a két komponens között fut. A régi színezés takarossága és a piros szabály miatt az f' színtelen és $c(f') \leq c(f)$. Az f' végpontjait összekötő F-beli út tartalmazza az f élet. Az 5. ismét alkalmazható ($g = f'$, $g' = f$): az F-be f helyett f'-t véve a kapott F' egy minimális költségű feszítőfa lesz, ami szavatolja, hogy az új színezés is takaros.

Nézzük a második állítást: tegyük fel, hogy van még egy f színtelen él. A színezés takarossága miatt a kék élek egy erdőt alkotnak. Ennek az erdőnek az összefüggő komponenseit a továbbiakban *kék fáknak* fogjuk nevezni. Ha f végpontjai ugyanabban a kék fában vannak, akkor a piros szabály alkalmazható arra körre, aminek az élei f és az f végpontjait összekötő (egyetlen) kék út élei. Ha f különböző kék fákat köt össze, akkor pedig a kék szabály működik; X legyen az egyik olyan fa csúcshalmaza, amihez f csatlakozik. (Ez utóbbi esetben nem biztos, hogy f fog színt kapni a következő lépésben.) A színezés tehát folytatható. □

Következmény: *Ha a piros-kék algoritmussal befestjük az összefüggő $G = (V, E)$ gráf minden élét, akkor a kék élek egy minimális költségű feszítőfa élei. Sőt, ez már akkor is igaz, amikor van $|V| - 1$ kék élünk (és esetleg van még színezetlen él).*

Bizonyítás: Az első állítás azonnal következik abból, hogy a végső színezés is takaros. A második pedig az elsőből: végül összesen $|V| - 1$ kék él lesz. Ha már van ennyi, akkor több nem keletkezhet. □

A piros-kék algoritmus tanulsága, hogy bárhol és bármikor is alkalmazzuk sorra a két szabályt, végül mindenképpen egy minimális költségű feszítőfát kapunk. A recept *helyessége* szempontjából tehát közömbös a sorrend. Mint később

látni fogjuk, a *hatékonyság* szempontjából viszont nem. A következő nevezetes algoritmusok a piros-kék módszer pontosított változatainak tekinthetők.

PRIM MÓDSZERE: Legyen s a G egy rögzített csúcsa. Minden egyes színező lépéssel az s-et tartalmazó F kék fát bővítjük. Kezdetben az F csúcshalmaza $\{s\}$, végül pedig az egész V. A következő kék élnek az egyik legkisebb súlyú élet választjuk azok közül, amelyek F-beli pontból F-en kívüli pontba mennek.

KRUSKAL MÓDSZERE: A következő befestendő f él legyen mindig a legkisebb súlyú színtelen él. Ha az f két végpontja ugyanazon kék fában van, akkor az él legyen piros, különben pedig kék.

Prim algoritmusa minden lépésben a kék szabályt használja; X-nek választható az F csúcshalmaza. A Kruskal-módszer a piros szabály szerint színez, amikor f-et pirosra festi. A szabály arra az egyszerű körre alkalmazható, aminek az élei f és a két végpontját összekötő kék út élei. Ha pedig f az F_1 és F_2 kék fákat köti össze, akkor X-et az F_1 csúcshalmazának választva teljesülnek a kék szabályban foglalt feltételek. Az előzőek szerint mindkét módszer végül egy minimális költségű feszítőfát ad.

Valamivel bonyolultabb stratégiát követ a Borůvka-módszer, aminek a működése menetekre osztható. Egy menetben több élet választunk ki, és azok mindegyikét kékre színezzük. Az egyszerűség kedvéért itt tegyük fel, hogy G-nek nincs két egyforma súlyú éle.

BORŮVKA MÓDSZERE: Minden egyes kék fához válasszuk ki a legkisebb súlyú belőle kimenő (színtelen) élet. Színezzük kékre a kiválasztott éleket.

Egy élet két fához is kiválaszthatunk; természetesen ekkor is csak egyszer kell kékre festeni. Ez a módszer is a piros-kék algoritmus specializált változta. Tegyük fel ugyanis, hogy az algoritmus valamelyik menetében az $f_1, f_2, \ldots, f_k$ éleket választotta ki. A számozás esetleges megváltoztatása árán feltehetjük, hogy az éleket súly szerint növekvően írtuk fel, azaz $c(f_1) < c(f_2) < \cdots < c(f_k)$. Jelöljön továbbá F_i egy olyan kék fát, amihez f_i-t választottuk, és legyen X_i az F_i csúcshalmaza. A Borůvka-módszer menete, amely az f_i éleket kékre festi, úgy tekinthető, hogy az X_i $(i = 1, 2, \ldots, k)$ halmazokra sorra alkalmazzuk a kék szabályt. Annyit kell csak észrevenni, hogy az $f_1, f_2, \ldots, f_{i-1}$ élek egyikének sincs pontja X_i-ben, ellenkező esetben ugyanis nem az f_i-t választottuk volna F_i-hez. Az f_i él színezése előtt X_i-ből nem megy ki kék él, így a kék szabály szerint f_i befesthető.

A módszer hatékonyan párhuzamosítható. Egy menetben az élek kiválasztása és színezése párhuzamosan végezhető. A menetek száma pedig igen kedvező korlát alatt marad.

Feladat: Mutassuk meg, hogy a Borůvka-módszer meneteinek száma legfeljebb $\log_2 n$; ennyi menet után már csak egy kék fa létezhet.

Az előzőekben bemutattuk három fontos feszítőfa-algoritmus lényegi lépéseit. A módszerek helyessége következik abból, hogy mindegyik a piros-kék algoritmus speciális esete. Ezután a Prim- és a Kruskal-algoritmusok hatékony megvalósításával foglalkozunk. Itt az adatszerkezetek játsszák a főszerepet.

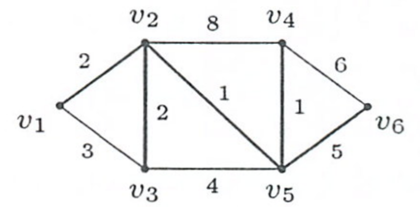
6.6.1. Prim módszere

Legyen $G = (V, E)$ egy összefüggő gráf, $c : E \to \mathbb{R}$ az élein értelmezett súlyfüggvény, és $V = \{1, \ldots, n\}$. R. C. Prim módszere egy mohó algoritmus, ami a piros-kék módszer részletesen kidolgozott változatának tekinthető. Az $s = 1$ csúcsból indulva a kék szabály alkalmazásával lépésről lépésre bővítgetjük az s-et tartalmazó kék fát. A kék éleket az F halmazban tároljuk. Végül F egy minimális költségű feszítőfa éleit tartalmazza. A kék élek kiválasztásához hasznos lesz még egy U változó, ami az aktuális kék fa csúcsait tartalmazza. Kezdetben U csak az 1-es csúcsból áll. A kék szabály alkalmazása úgy történik, hogy kiválasztjuk azon élek közül a legkisebb költségűt, melyek egyik végpontja U-beli, a másik pedig $V \setminus U$-beli. Vagyis mondhatjuk, hogy az U-hoz legközelebbi „külső" csúcsot jelöljük ki. Ezt a csúcsot U-ba, a kiválasztott élet pedig a fokozatosan növekvő kék fába tesszük.

```
procedure Prim (G: gráf; var F: élek halmaza);
    var
        U: csúcsok halmaza;
        u, v: csúcsok;
    begin
        F := ∅;
        U := {1};
        while U ≠ V do begin
(*)         legyen (u, v) egy legkisebb súlyú olyan él,
            melyre u ∈ U és v ∈ V \ U;
            F := F ∪ {(u, v)};
            U := U ∪ {v}
        end
    end
```

A Prim-algoritmus helyessége, vagyis hogy végül F a G gráf egy minimális költségű feszítőfájának éleit tartalmazza, következik a piros-kék algoritmus ha-

sonló tulajdonságából. A következő rajzon egy súlyozott élű irányítatlan gráf látható. Az egyetlen minimális költségű feszítőfájának az éleit megvastagítottuk.



Ha Prim módszerével a v_1 csúcstól indulunk, akkor a pontok a $v_1, v_2, v_5, v_4, v_3, v_6$ sorrendben kerülnek be az U halmazba. Ennek megfelelően F-be először a (v_1, v_2) élet vesszük, amit (v_2, v_5), (v_5, v_4), (v_2, v_3) és végül (v_5, v_6) követ.

Ezután a Prim-algoritmus megvalósításának finomabb részleteivel foglalkozunk.

Naiv implementáció:

Tegyük fel, hogy a gráf az élsúlyokat tartalmazó C adjacencia-mátrixával[5] adott. A $(*)$ sor megvalósításához minden lépésben ki kell tudjuk választani az épp aktuális U és $V \setminus U$ halmazok közt futó legkisebb súlyú élek egyikét. Az összes él megvizsgálása nyilván nagyon sok időt venne igénybe. Ezért minden $V \setminus U$-beli csúcshoz tároljuk, hogy milyen messze van az U halmaztól, vagyis a belőle az U halmazba futó élek közül egy minimálisnak a súlyát. Tartsuk még nyilván ezen élek U-beli végpontját is. Erre szolgál a következő két tömb:

$$\text{KÖZEL}[i] = \begin{cases} * & \text{ha } i \in U \\ \text{egy az } i\text{-hez legközelebbi } U\text{-beli csúcs} & \text{ha } i \in V \setminus U \end{cases}$$

$$\text{MINSÚLY}[i] = \begin{cases} * & \text{ha } i \in U \\ C[i,j] & \text{ha KÖZEL}[i] = j \neq * \end{cases}$$

A következő kék él az $(i, \text{KÖZEL}[i])$ élek közül kerül majd ki. Ezért ezeket az éleket *kékes* éleknek nevezzük. Kezdetben $U = \{1\}$. Ennek megfelelően a kezdőértékek:

[5]Mint ahogy a Dijkstra-algoritmusnál, itt is feltesszük, hogy $C[i, j] = \infty$, ha (i, j) nem éle G-nek.

$$\text{KÖZEL}[i] := \begin{cases} * & \text{ha } i = 1 \\ 1 & \text{ha } i \neq 1 \end{cases}$$

$$\text{MINSÚLY}[i] := \begin{cases} * & \text{ha } i = 1 \\ C[i,1] & \text{ha } i \neq 1 \end{cases}$$

Ezek után a $(*)$ sor teendőit így valósíthatjuk meg:

- *A következő kék él kiválasztása:* megkeressük a MINSÚLY[] tömb minimumát, vagyis a legrövidebb kékes él hosszát. Mondjuk ez legyen k-nál. A minimumkeresés költsége: $O(n)$ elemi lépés. Nyilvánvaló, hogy a $V \setminus U$-beli csúcsok közül k egyike az U-hoz legközelebbieknek. Tehát az (u, v) él lehet a (KÖZEL$[k], k$) él; ezt fogjuk F-be tenni, k-t pedig U-ba. Ennek megfelelően a két tömb k-hoz tartozó értékét $*$-ra kell állítanunk: MINSÚLY$[k]$:= KÖZEL$[k]$:= $*$. E teendők időigénye $O(1)$).

- *A két tömb felfrissítése:* (Erre azért van szükség, mert az U halmaz bővült k-val. Így elképzelhető, hogy valamely $V \setminus U$-beli csúcshoz tartozó tömbértékeket meg kell változtatni, mert a csúcs közelebb van k-hoz, mint bármely régi U-beli csúcshoz.) Ehhez a $C[k, i]$ és a MINSÚLY$[i]$ értékeket $(i \in V \setminus U)$ kell összevetni. Ez $O(n)$ költséggel megtehető a következő kódrészlet végrehajtásával (minden $i \in V \setminus U$ pontra):

```
if KÖZEL[i] ≠ * and C[k,i] < MINSÚLY[i] then begin
   KÖZEL[i] := k;
   MINSÚLY[i] := C[k,i]
end
```

Mindent egybevetve, a **while**-ciklus belsejének végrehajtása $O(n)$ időt tesz ki. Mivel a ciklus n-szer fut le, az algoritmus összidőigénye naiv implementáció esetén $O(n^2)$.

Kupacos-éllistás implementáció:

Ha a gráfunk viszonylag ritka, vagyis az élek száma jóval kevesebb, mint n^2, akkor érdemes éllistát használni a gráf tárolására, és az algoritmust a következőképp megvalósítani.

A **while**-ciklus magjának minden egyes lefutásakor azon élek közül kell a minimális súlyút kiválasztanunk, melyek egyik végpontja U-ban, a másik $V \setminus U$-ban

van. Az U halmaz bővítése után lesznek újabb ilyen tulajdonságú élek, ezeket hozzá kell venni a többihez (nem törődünk az esetleg ottmaradó most már U-n belüli élekkel). Vagyis MINTÖR és BESZÚR műveleteket kell hatékonyan implementálnunk, amihez a kupac adatszerkezet lesz segítségünkre. Tehát építsünk és tartsunk fenn kupacot az U és $V \setminus U$ közti élekből (rendezés élsúly szerint). Ez kezdetben csak az 1-es csúcsból kiinduló éleket tartalmazza. Ezek után a ciklus belsejében az (u, v) él kiválasztása néhány MINTÖR művelettel megvalósítható. Azért néhánnyal és nem feltétlenül eggyel, mert nem foglalkozunk külön azoknak az éleknek a kupacból való törlésével, melyeknek egy csúcs hozzávétele után mindkét végpontja U-beli lesz. Így a MINTÖR eredményeképp kaphatunk ilyen élet is. Ilyenkor újabb MINTÖR-rel próbálkozunk, míg nem találunk U-ból kimenő élet. A következő kék él meghatározása után BESZÚR műveletekkel hozzávesszük a kupachoz a kiválasztott v csúcsból a $V \setminus U$ halmazba menő éleket.

A kupac mérete sosem haladja meg e-t. Ezért a kezdeti kupacépítés legfeljebb $O(e)$, az egyes műveletek végrehajtása pedig $O(\log e)$ időt vesz igénybe. Összesen kevesebb, mint e darab BESZÚR és legfeljebb e darab MINTÖR műveletet végzünk. Tehát az algoritmus költsége itt $O(e \log e)$. Mint ahogy azt megjegyeztük, a kupacos-éllistás implementációt kevés élet tartalmazó gráfoknál érdemes használni.

Johnson módszere:

A következő – D. B. Johnsontól származó – elképesztően erőteljes implementáció merít a két előző megközelítés gondolataiból. Nyilvántartjuk egyrészt a kékes éleket, mint ahogy azt a naiv implementációnál tettük; másrészt kupacot használunk a minimum kiválasztására. A $j \in V \setminus U$ csúcshoz tároljuk egy belőle kiinduló kékes él adatait. Erre a célra egy $\boxed{\; j \;\mid\; i \;\mid\; c(i,j) \;}$ szerkezetű cellát állítunk össze. Itt i egy a j-hez legközelebbi U-beli csúcs, a $c(i, j)$ pedig az (i, j) kékes él súlya. A cellákat a c-érték mint kulcs szerint d-kupacban tartjuk. Először csak az 1-es csúcsból kimenő élek végpontjainak cellái lesznek a kupacban. Kezdetben tehát a kupac a $\boxed{\; j \;\mid\; 1 \;\mid\; c(1,j) \;}$ alakú cellákból áll, ahol $(1, j)$ éle G-nek.

A kupac létrehozásának (kupacépítés) a költsége $O(n)$. A Prim-algoritmus $(*)$ során a kékre festendő (u, v) él meghatározása egy MINTÖR-rel megtehető. Ezután – a naiv változathoz hasonlóan – gondoskodnunk kell a kékes élek halmazának a frissítéséről. Ehhez meg kell vizsgálnunk a v éllistáján szereplő (v, w) éleket. (Itt v az U halmazba éppen bekerült csúcs, ezért a belőle kimenő élek szóba jöhetnek mint kékes élek.) Ha most $w \in U$, akkor nincs további teendőnk ezzel az éllel. Ha $w \notin U$, akkor két eset lehetséges. Előfordulhat egyfelől, hogy w-ből eddig nem ment él U-ba. Ekkor (v, w) nyilván egy kékes él lesz. Ennek megfelelően egy BESZÚR művelettel a $\boxed{\; w \;\mid\; v \;\mid\; c(v,w) \;}$ cellát a kupacba tesszük. Másfelől

az is lehetséges, hogy w-ből már megy ki kékes él, de a (v, w) él ennél rövidebb. Ezt onnan látjuk, hogy a w cellája a kupacban van, és a benne levő súly nagyobb, mint $c(v, w)$. Ekkor a cellát átírjuk, aminek az eredménye $\boxed{w \mid v \mid c(v,w)}$ lesz. A módosított cellában a kulcs értéke csökkent, tehát egy FOGYASZT segítségével illeszthetjük vissza a kupacba. Ezek a módosítások a G éleihez köthetők. Egy él kapcsán legfeljebb egyszer módosítunk; mégpedig akkor, amikor egyik végpontja éppen bekerül U-ba, és a másik nincs U-ban. A fogyasztások száma tehát legfeljebb e. A beszúrások száma pedig legfeljebb $n-1$, hiszen egy csúcs cellája legfeljebb egyszer kerül a kupacba. Figyelembe véve, hogy a kupacban legfeljebb $n-1$ cella van, a műveletek költsége összesen $O(n + nd\log_d n + n\log_d n + e\log_d n)$. Az első tag a kupacépítés, a második az $n-1$ darab MINTÖR, a harmadik a beszúrások, az utolsó pedig a fogyasztások lépésszáma. A formula hasonlít a Dijkstra-módszer d-kupacos változatának elemzésénél kapott kifejezéshez. Hasonló a következtetés is, amit levonhatunk belőle. Tegyük fel, hogy G nem túl ritka, mondjuk $n^{1,5} \le e$. Legyen ekkor $d = \lceil e/n \rceil$. Nyilvánvaló, hogy $d \ge \sqrt{n}$, amiből $\log_d n \le 2$. Ezt figyelembe véve

$$\begin{aligned} O(n + nd\log_d n + n\log_d n + e\log_d n) &= O(n + nd + n + e) \\ &= O(n + n\cdot e/n + n + e) = O(e). \end{aligned}$$

Viszonylag sűrű gráfok esetén tehát a kupacműveletek összköltségére lineáris korlátot kaptunk.

Maradt még egy elvarratlan szál. A w csúccsal a kezünkben gyorsan el kell tudnunk dönteni, hogy az már/még a kupacban van-e; ha igen, akkor meg kell találni a celláját. Ezt egy MERRE[2 : n] tömb segítségével egyszerűen megtehetjük. Legyen MERRE[w] $= \infty$, ha w cellája még nincs a kupacban, és legyen $*$, ha w már U-ba került. Ha pedig a w cellája a kupacban van, akkor MERRE[w] legyen egy mutató erre a cellára. A tömb segítségével a w-vel kapcsolatos döntés konstans időben meghozható. A tömb karbantartásához szükséges munka $O(e)$. Ez azért igaz, mert egy kupacművelet után állandó mennyiségű munkával aktualizálható a tömb; a kupacműveletek száma pedig $O(e)$. A Johnson-implementáció tehát *lineáris költségű*, ha G viszonylag sűrű ($n^{1,5} \le e$) gráf.

6.6.2. Kruskal módszere

Legyen továbbra is $G = (V, E)$ egy összefüggő gráf, $c : E \to \mathbb{R}$ az élein értelmezett súlyfüggvény, és $V = \{1, \ldots, n\}$. Kruskal algoritmusa, mely G egy minimális költségű feszítőfáját konstruálja meg, szintén egyfajta mohó stratégiát követ. Míg Prim módszerénél egyetlen kék fát növesztettünk, itt több helyen kezdjük el építgetni, és kapcsoljuk fokozatosan össze a még diszjunkt darabokat. A piros éleket figyelmen kívül hagyva elmondhatjuk, hogy a kék élek F halmazát bővítjük a

legkisebb súlyú olyan éllel, amely az eddig kiválasztott élekkel együtt nem alkot kört. Nyilván egy ilyen él két diszjunkt kék fát köt össze. A kiinduló helyzetben n egypontú fánk van. Minden lépésben a legkisebb súlyú, kört nem okozó él hozzávételével a kék komponensek száma eggyel csökken. Így $n-1$ él kiválasztása után egyetlen kék fa lesz, nevezetesen G egy minimális költségű feszítőfája. Nagy vonalakban ez így írható le (H kezdetben a gráf összes éleinek halmaza):

procedure Kruskal (G: gráf; **var** F, H: élek halmaza);

begin

 $F := \emptyset$; $H := E$;

 while $H \neq \emptyset$ **do begin**

 Töröljük a H minimális súlyú (v, w) élét.

 Ha $F \cup \{(v, w)\}$-ben nincs kör

 (azaz a v, w pontok különböző kék fákban vannak), akkor

 $F := F \cup \{(v, w)\}$

 end

end

Ha tehát a (v, w) él kört eredményez, akkor eldobjuk (piros él), ha pedig nem, akkor bevesszük a feszítőfa (kék) élei közé. Mint korábban láttuk, a Kruskal-módszer is a piros-kék algoritmus egy változata. Érvényes tehát a következő:

Állítás: *A Kruskal-algoritmus eredményeként végül F a G gráf egy minimális költségű feszítőfájának éleit tartalmazza.* □

Az előző példa gráfjára alkalmazva a Kruskal-algoritmus először a (v_2, v_5) és a (v_4, v_5) éleket választja valamilyen sorrendben; utána a minimális feszítőfa 2 súlyú élei jönnek és végül a (v_5, v_6) él.

Ezután itt is a hatékony implementáció részleteinek kimunkálásával foglalkozunk. Feltesszük, hogy G éllistával adott. A minimális élek kiválasztására az egyik lehetséges megoldás, hogy a G éleiből súly szerint kupacot építünk (költség $O(e)$). Ez esetben a minimális súlyú él kiválasztása, és a kupac-tulajdonság helyreállítása $O(\log e)$ időbe kerül. Ha tehát H-t kupacként valósítjuk meg, akkor a vele kapcsolatos összköltség $O(e \log e)$ lesz (kupacépítés és legfeljebb e MINTÖR). Ugyanebben a nagyságrendben maradunk, ha az éleket előzetesen rendezzük súly szerint, és az eredményül kapott rendezett lista lesz a H.

Már csak egy fontos kérdés maradt hátra: hatékony megoldás kellene annak az eldöntésére, hogy a kiválasztott (v, w) él az F eddigi éleivel kört alkot-e. E döntés eredményétől függően lesz az él piros vagy kék. Tartsuk nyilván az aktuálisan egy kék fába tartozó csúcsokat mint halmazokat. Kétféle műveletre lesz szükségünk.

Amikor a két különböző fát összekötő élet hozzávesszük F-hez (vagyis kékre színezzük), akkor a megfelelő két halmaz unióját kell képeznünk, hiszen ezekből így egyetlen összefüggő komponens lesz. Ahhoz pedig, hogy eldöntsük egy élről, hogy két különböző kék fát köt-e össze, az egyes végpontokról meg kell tudnunk mondani, hogy melyik halmazban vannak. Ezen műveletek hatékony megvalósítását szolgálja a következő adatszerkezet.

6.6.3. Az UNIÓ-HOLVAN adatszerkezet

Legyen adott egy véges S halmaz. Ennek egy felosztását – szakszóval: partícióját – szeretnénk tárolni. Emlékeztetőül: a nem üres $U_1, \ldots, U_k \subseteq S$ halmazok az S egy partícióját alkotják, ha $\cup_{i=1}^{k} U_i = S$ és $i \neq j$ esetén $U_i \cap U_j = \emptyset$. Az U_i halmazok tehát hézagtalanul és egyrétűen lefedik S-et. Két műveletünk van. Egy $x \in S$ esetén meg kell tudnunk mondani, hogy az x melyik U_j részhalmazba esik. A másik művelet a partíció két osztályának az egyesítése. Valamivel formálisabban:

> Adott egy n elemű S halmaz, és ennek bizonyos $U_1, \ldots, U_m$ részhalmazai, melyekre $U_i \cap U_j = \emptyset$ $(i \neq j)$ és $U_1 \cup \ldots \cup U_m = S$ (vagyis az U_j részhalmazok S egy partícióját adják).
>
> *Műveletek:*
>
> UNIÓ$(U_i, U_j) = (\{U_1, \ldots, U_m\} \cup \{U_i \cup U_j\}) \setminus \{U_i, U_j\}$ (az U_i, U_j halmazokat $U_i \cup U_j$ helyettesíti).
>
> HOLVAN(v) eredménye (itt $v \in S$) annak az U_i halmaznak a neve, amelynek v eleme.

Egy UNIÓ hatására a felosztás durvább lesz, minthogy eggyel csökken a komponensek száma. Az UNIÓ-HOLVAN adatszerkezet nevezetes alkalmazásainál több UNIÓ és HOLVAN műveletből álló lépéssor végrehajtására van szükség *abból a kezdőhelyzetből indulva, amelyben mindegyik U_j egyelemű*. Mi is élünk ezzel a feltevéssel.

Kruskal algoritmusában tényleg valami ilyesmire van szükség: legyen ugyanis $S = V(G)$, az U_j halmazok pedig az F által meghatározott kék gráf fáinak a csúcshalmazai. Az éppen vizsgált (v, w) él az F-hez adva pontosan akkor nem eredményez kört, ha a végpontok különböző komponensben vannak, azaz HOLVAN$(v) \neq$ HOLVAN(w). A körmentesség tehát két HOLVAN kérdéssel ellenőrizhető. Miután a (v, w) élet F-be tettük, egyesítenünk kell a v-t és a w-t tartalmazó kék fákat. Ezt egy UNIÓ művelettel érhetjük el. Az induló feltétel is teljesül, ugyanis kezdetben n darab egypontú fánk van.

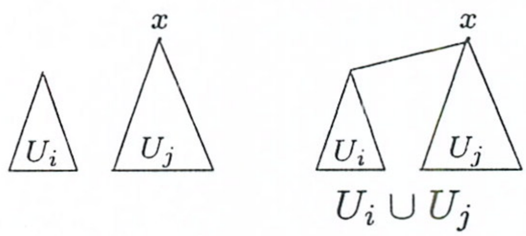
Célunk tehát az UNIÓ és a HOLVAN műveletek hatékony implementálása. A legegyszerűbb út, ha felveszünk egy n hosszú tömböt, és S minden eleméhez

tároljuk, hogy aktuálisan melyik részhalmazban van. Ekkor egy HOLVAN végrehajtása $O(1)$, egy UNIÓ pedig n-nel arányos időt vesz igénybe. Az előbbinél a tömb egy elemét kell lekérdeznünk, az utóbbinál pedig végig kell mennünk a tömbön, és a megfelelő két halmaznevet azonosra változtatnunk. Ennél azonban létezik lényegesen hatékonyabb megoldás is.

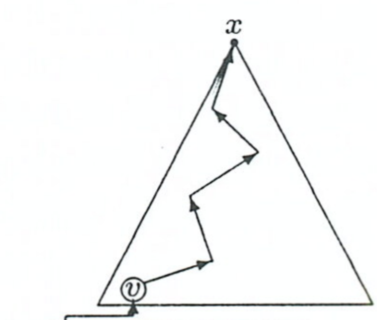
Implementáció fákkal

Egy U_j részhalmaznak egy gyökeres, felfelé irányított fa felel meg. U_j elemeit a fa csúcsaiban tároljuk, egy szülőmutatóval együtt (ami egy nullértéket tartalmaz a gyökérnél). Egy részhalmaz *neve* legyen az őt ábrázoló fa gyökere. A gyökérben nyilvántartjuk még a fa méretét is. Van továbbá egy S elemeivel indexelt tömbünk. Ennek a v indexű ($v \in S$) eleme egy mutató a v fabeli előfordulására. A műveletek megvalósítása:

- **UNIÓ:** $U_i \cup U_j$ fáját a következőképpen készítjük el:
 Tegyük fel, hogy $|U_i| \leq |U_j|$. Ekkor az U_j fa x gyökeréhez gyermekként hozzákapcsoljuk U_i gyökerét.



- **HOLVAN:** A $v \in S$ elemet tartalmazó részhalmaz nevét, azaz a megfelelő fa gyökerét a szülőkhöz menő mutatók végigkövetésével találhatjuk meg.



Tegyük fel, hogy az UNIÓ hívásakor az U_i és U_j halmazok a gyökerükkel adottak (ez a feltevés valós a Kruskal-módszernél). Az UNIÓ költsége ekkor nyilván $O(1)$, hiszen a gyökereknél adminisztráljuk az egyes fák méretét, és csak a kisebbik fa gyökerének szülőmutatóját kell átállítanunk. Vegyük észre, hogy amikor egy v csúcs új gyökér alá kerül, akkor egy szinttel lesz távolabb a gyökértől, míg az új fájának a mérete legalább az eredeti duplájára változik. Ezért egy csúcs legfeljebb $\log_2 n$-szer kerülhet új gyökér alá, így a szintszáma[6] legfeljebb $\log_2 n$ lehet. Ebből adódik, hogy a HOLVAN költsége $O(\log n)$. Ha nem vigyáztunk volna, hogy mindig a kisebb fát kapcsoljuk a nagyobbhoz, akkor a fa mélységére nem tudnánk ilyen korlátot adni; a HOLVAN költsége akár n-nel arányos is lehetne. Vegyük észre azt is, hogy ebben az elemzésben használtuk a kezdeti partícióval kapcsolatos feltételezést.

Kruskal algoritmusában egy él mindkét végpontjáról egyszer kérdezzük le, hogy melyik komponensben van. Összesen $n - 1$ élet választunk be a feszítőfába. Ez tehát $2e$ HOLVAN, és $n - 1$ UNIÓ műveletet jelent. Ezek időigénye $O(e \log n + n) = O(e \log n)$, vagy ami ugyanaz: $O(e \log e)$. Ebbe belefér a kiindulási helyzetet jelentő n egypontú fa kialakításának az $O(n)$ költsége is. Ugyancsak $O(e \log e)$ az összköltsége a minimális súlyú élek kiválasztásával kapcsolatos munkáknak, akár kupacot használunk, akár előzetesen rendezzük az éleket.

Következmény: *A Kruskal-algoritmus költsége* $O(e \log e)$. □

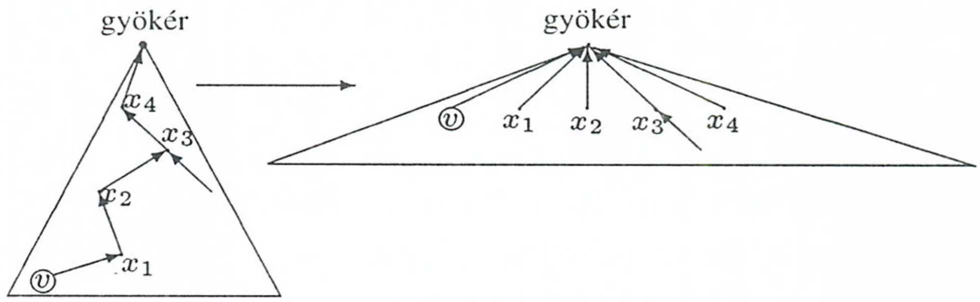
Az UNIÓ-HOLVAN adatszerkezetet eddig jórészt afféle „individualista" szemlélettel néztük, amennyiben az egyes műveletek egyetlen végrehajtásának az idejére adódó korlátokkal foglalkoztunk. Valójában az érdekes alkalmazásoknál nem egy, hanem egy egész sor UNIÓ-t és HOLVAN-t kell végrehajtanunk. Ilyenkor érdemes lehet a hosszabb műveletsorok idejét javítani. Az elgondolás hasonlít az S-fáknál és az önszervező tábláknál látottakhoz: amikor egy HOLVAN kérdést megválaszolunk, akkor némi további munkával igazítunk az éppen elért fa alakján azt remélve, hogy ez a befektetés később megtérül. Ebben nem is csalatkozunk.

A HOLVAN gyorsítása: útösszenyomás

Az UNIÓ és a HOLVAN műveletek közül az utóbbi a költséges; ennél érdemes takarékoskodni az idővel. A HOLVAN(v) időigénye a v-től a fája gyökeréig vivő út hosszával arányos. Ha tehát gondoskodunk arról, hogy az elemek viszonylag közel legyenek a gyökérhez, akkor jobb időket kaphatunk. Egy ilyen ötlet az *útösszenyomás*. Ennek lényege, hogy HOLVAN(v) meghívása esetén v fájában a v-től a gyökérhez vezető út minden pontját közvetlenül a gyökér alá csatlakoztatjuk át.

[6]Egy csúcs *szintszáma* a tőle a gyökérig vezető úton levő élek száma.

Ezt legegyszerűbben úgy tehetjük meg, hogy még egyszer végigmegyünk ezen az úton, és átállítjuk a szülőmutatókat. Így egy kis pluszmunkával javítunk a fa alakján. Az x_i csúcsok és a leszármazottaik közelebb kerülnek a gyökérhez.



Az útösszenyomást alkalmazó implementáció időigényének a becslése meglehetősen bonyolult; csak az eredményt ismertetjük.

Tétel: *Legyen $|S| = n$, és tegyük fel, hogy kezdetben mindegyik U_j egyelemű. Ha egy olyan utasítássorozatot hajtunk végre (útösszenyomással), melyben $n-1$ UNIÓ és $m \geq n-1$ HOLVAN szerepel, akkor ennek az időigénye $O(m\alpha(m))$.*

A korlátban szereplő $\alpha : \mathbb{Z}^+ \to \mathbb{Z}^+$ függvény az *inverz Ackermann-függvény*. Az α cirkalmas definíciójától eltekintünk. Csak annyit jegyzünk meg róla, hogy $\alpha(m)$ a végtelenhez tart, ha $m \to \infty$, ezt azonban a képzeletet jócskán próbára tevő lassúsággal teszi. Lassabban nő például, mint a logaritmus k-szori önmagába helyettesítésével adódó $\log\log\cdots\log m$ függvény ($k \in \mathbb{Z}^+$ tetszőleges). Igaz továbbá, hogy $\alpha(m) \leq 4$, ha $m < 2^{65536}$. A praktikus méretű feladatoknál tehát $\alpha(m)$ állandónak (≤ 4) tekinthető.

A Kruskal-algoritmussal kapcsolatos teendők két csoportba sorolhatók. Az egyik csoportot a minimális élek választásai jelentik, a másikat pedig az UNIÓ-HOLVAN kezelése. Az első csoport összköltségére $O(e \log e)$ korlátot kaptunk. Gyakran előfordul azonban, hogy ezek a rendezéssel kapcsolatos teendők $O(e)$ lépésben is megoldhatók. Ilyen helyzet például, amikor az élsúlyok kis egészek, és ezért az élek a radix-módszerrel lineáris időben rendezhetők. Az is megeshet, hogy a bemenetben már eleve rendezetten kapjuk az éleket. Ezekben az esetekben az általánosnál jobb korlátot nyerünk.

Alkalmazás: *Ha az élek rendezésével kapcsolatos teendők $O(e)$ időben megoldhatók, akkor a Kruskal-algoritmus $O(e\alpha(e))$ időben megvalósítható.*

Bizonyítás: A második csoportba tartozó teendőkre alkalmazható a tétel: $n-1$ UNIÓ-t és legfeljebb $2e$ HOLVAN-t hajtunk végre. A feltétel szerint az összköltség becslésében a domináns tag az e műveletekre vonatkozó $O(e\alpha(e))$ lesz. □

6.6.4. Megjegyzések

1. Az általunk szemügyre vett algoritmusok (Borůvka, Prim és Kruskal módszerei) az élsúlyokkal csak összehasonlításokat végeznek. Nem vizsgálják például a bitjeiket, nem adják össze őket, stb. Izgalmas nyitott kérdés, hogy van-e az ilyen értelemben összehasonlítás alapú algoritmusok között lineáris költségű. Közel lineáris módszerek léteznek. A Borůvka-módszernek például van $O(e \log \log n)$ költségű megvalósítása (A. C. Yao, 1975). A lineáris időkorlát elérhető véletlen választásokkal (D. R. Karger, P. N. Klein, R. E. Tarjan, 1994). Módszerük várhatóan $O(e)$ lépésben talál egy minimális költségű feszítőfát.

M. Fredman és D. E. Willard (1990) módszere lineáris idejű ugyan, de kulcsmanipulációs jellegű, amennyiben az élsúlyok bitjeivel is végez számításokat.

Érdekes kapcsolódó eredmény, hogy ha adott G és annak egy F élhalmaza, akkor lineáris időben eldönthető, hogy F minimális költségű feszítőfa-e (Komlós János 1985, V. King 1993).

2. A minimális feszítőfák problémájának több figyelemre méltó rokona és változata van. Ezekből most két feladatban mutatunk mintát. Az első a probléma *online* változatáról szól. Tegyük fel, hogy a G éleit egyesével kapjuk valamilyen sorrendben, és szeretnénk gyorsan meghatározni az eddig látott élekből álló gráf egy minimális költségű feszítő erdejét (amelynek az összefüggő komponensei ugyanazok, mint az eddigi élekből álló gráfnak). Evégből tegyük fel, hogy az eddigi élek közül már kiválasztottunk egy kék feszítő erdőt, és jön a következő f él. A recept egyszerű: színezzük az f élet kékre; ha emiatt kialakul egy kék kör, akkor abból töröljünk egy maximális súlyú élet.

Feladat: Mutassuk meg, hogy az előző algoritmus helyes; ha az üres gráfból indulva sorra alkalmazzuk az $f_1, f_2, \ldots, f_i$ élekre, akkor a kék élek halmaza az $f_1, f_2, \ldots, f_i$ élekből álló G_i gráf egy olyan minimális költségű feszítő erdeje lesz, amelynek ugyanannyi összefüggő komponense van, mint G_i-nek. Speciális esetként: ha G összefüggő, akkor végül – amikor már G minden élét láttuk – egy minimális feszítőfát kapunk. (Használjunk i szerinti indukciót.)

Egy feszítőfa költségét úgy definiáltuk, hogy az a benne szereplő élek súlyainak az összege. Bizonyos esetekben másféle költségszámítás is értelmes lehet. Például egy fa költségeként értelmezhetjük a legnagyobb súlyú élének a súlyát[7].

Feladat: Mutassuk meg, hogy a piros-kék algoritmus minimális feszítőfát ad akkor is, ha így értelmezzük egy fa költségét.

[7] Ez a költségfogalom merül fel, amikor az élsúlyok a csúcsok közötti kapcsolatok valamiféle megbízhatatlanságát mérik. A nagyobb súlyú él tehát kevésbé megbízható. A minimális feszítőfa ilyenkor egy olyan összeköttetés-rendszert jelent, amiben a leggyengébb láncszem a lehető legmegbízhatóbb.

6.7. Maximális párosítás páros gráfokban

A lovagok és udvarhölgyek problémája egyike volt bevezető példáinknak (az első fejezetben). Itt ennek a kérdésnek az általánosításával, a *párosítási feladattal* foglalkozunk. A párosítási feladatnak több fontos változata van. Ezek közös magja, hogy egy gráfban minél nagyobb olyan élrendszert (szakszóval: *párosítást*) keresünk, amelyben semelyik két élnek sincs közös végpontja. A *nagy* az előző mondatban utalhat a kiválasztott élek számára vagy – ezen túlmenően – a kiválasztott élek súlyainak összegére.

Párosítási feladatként fogalmazható meg egy sereg erőforrás-kiosztási probléma. Például tegyük fel, hogy adottak gépek és munkák véges halmazai: G és M. Egy gép egy időszakban csak egy munkával tud foglalkozni. Ismert továbbá, hogy egy gép mely munkák elvégzésére alkalmas. Szeretnénk egy időszakban minél jobban kihasználni a gépeket: minél több géphez akarunk munkát rendelni úgy, hogy ne sértsük meg az előző feltételeket. A problémának megfelelő gráf ponthalmaza $G \cup M$. A $g \in G$ gépet akkor köti él az $m \in M$ munkához, ha g képes az m elvégzésére. A megoldások a gráf maximális élszámú párosításai.

A következőkben a párosítási feladat legegyszerűbb esetével foglalkozunk, amikor is a szóban forgó gráf páros, és nincsenek élsúlyok (vagyis minden élsúly 1). Lássuk először az idevágó fogalmakat:

Definíció (páros gráf): *A $G = (V, E)$ gráfot* párosnak *nevezzük, ha V csúcshalmaza felosztható két diszjunkt részre – V_1-re és V_2-re – úgy, hogy minden él ezen két halmaz között fut, vagyis $(x, y) \in E$ esetén $x \in V_1$ és $y \in V_2$ vagy fordítva.*

Ha egy $G = (V, E)$ páros gráfot úgy adunk meg, hogy $G = (V_1, V_2; E)$, akkor V_1 és V_2 a fenti tulajdonságú partíciója a V-nek.

Definíció (párosítás): *Legyen $G = (V, E)$ egy tetszőleges gráf. Az E élhalmaz $E' \subseteq E$ részhalmaza G egy* párosítása, *ha a $G' = (V, E')$ gráfban minden pont foka legfeljebb egy.*

A párosításban szereplő éleket, illetve a párosítás által fedett pontokat (vagyis melyek foka G'-ben egy) *párosítottnak* fogjuk hívni.

Definíció (maximális párosítás): *A G gráf egy E' párosítása* maximális, *ha G minden E'' párosítására $|E''| \leq |E'|$.*

Ezek után megfogalmazhatjuk pontosan a feladatot:

A probléma: Adott egy $G = (V_1, V_2; E)$ páros gráf. Határozzuk meg G egy maximális párosítását.

6.7.1. A magyar módszer

Az itt bemutatásra kerülő algoritmus első változatát König Dénes[8] közölte 1916-ban. A módszert König Dénes és a súlyozott feladatot megoldó Egerváry Jenő tiszteletére nevezik világszerte magyar módszernek. A következő két fogalom elvezet bennünket az algoritmus lényegéhez.

Definíció (alternáló út): *Legyen G egy tetszőleges gráf, és E' a G egy párosítása. Egy G-beli utat* E'-alternáló útnak *hívunk, ha felváltva tartalmaz párosított és nem párosított éleket.*

Ha a G gráf egy adott E' párosításához találunk olyan P alternáló utat, amely két különböző párosítatlan csúcsot köt össze, akkor ez az út párosítatlan éllel kezdődik és fejeződik be. P-nek tehát eggyel kevesebb párosításbeli éle van, mint nem párosításbeli. Ezért az E'-nél jobb párosítást kapunk, ha elhagyjuk belőle a P út eddig párosított éleit, de hozzávesszük az eredetileg párosítatlanokat. Az ilyen tulajdonságú utat, melynek segítségével javítani tudunk egy meglévő párosításon, javító útnak nevezzük. Formálisan:

Definíció (javító út): *Legyen E' a $G = (V, E)$ gráf egy párosítása. Ekkor egy olyan E'-alternáló út, melynek mindkét végpontja párosítatlan,* E'-re nézve javító út, *vagy röviden* E'-javító út.

Arra jutottunk, hogy ha a G gráfban adott egy E' párosítás és egy P út, mely E'-javító út, akkor az $E'' = E' \oplus P$ egy az E'-nél nagyobb párosítás. Itt $\oplus$ a halmazok körében értelmezett szimmetrikus differencia műveletét jelöli (a P utat ilyenkor élek halmazaként fogjuk fel). Két halmaz szimmetrikus differenciája az a halmaz, melynek elemei a két halmaz valamelyikében szerepelnek, de mindkettőben nem:

Jelölés (szimmetrikus differencia): *Legyenek H_1, H_2 tetszőleges halmazok.*

$$H_1 \oplus H_2 = (H_1 \setminus H_2) \cup (H_2 \setminus H_1).$$

[8] König Dénes (1884-1944) a Budapesti Műszaki Egyetem tanára volt. A *Theorie der endlichen und unendlichen Graphen* című, 1936-ban Lipcsében megjelent munkája az első könyv, amely rendszerbe foglalva tárgyal gráfokkal kapcsolatos ismereteket. A gráfelmélet ettől kezdve tekinthető önálló tudományos témának. A mű fontosságát jelzi, hogy 1950-ben reprint kiadása született az Egyesült Államokban, 1990-ben pedig angolra fordították. Tudomásunk szerint ő hirdetett meg először a történelemben gráfelméleti tárgyú egyetemi előadást; ez az 1927/28-as tanévben történt.

A tudósportré mosolygós vonásaként említhetjük a *Mathematikai mulatságok* címmel 1902-ben és 1905-ben megjelent két könyvecskéjét. Ezekben a gyűjteményekben érdekes matematikai rejtvényeket adott közzé. A feladatok szépsége, a magyarázatok áttetsző tisztasága teszi őket nagyszerű olvasmánnyá. A két füzetet 1992-ben ismét kiadták (Typotex Kft.).

Ezek alapján az algoritmus úgy néz ki, hogy kiindulunk valami könnyen szerzett párosításból (ez lehet üres is), és minden lépésben az aktuális párosításra nézve keresünk egy javító utat. A javító út mentén egy nagyobb párosítást kapunk. De mi van akkor, ha már egy párosításhoz nem találunk javító utat? A következő tétel pont azt mondja ki, hogy ekkor nyugodtan megállhatunk, mert az aktuális párosításunk már maximális.

Tétel: *Legyen $G = (V, E)$ egy tetszőleges gráf és E' egy párosítása. Ha E'-re nézve nincs javító út G-ben, akkor E' a G egy maximális párosítása.*

Bizonyítás: Legyen M a G egy tetszőleges maximális párosítása. Vizsgáljuk meg a $G' = (V, E' \oplus M)$ gráfot. Minden pontra mindkét párosításban legfeljebb egy él illeszkedhet, ezért G'-ben minden pont foka legfeljebb kettő. Ez azt jelenti, hogy G' komponensei utak és körök, amelyek felváltva tartalmaznak E'-beli és M-beli éleket. Így a körök csak páros hosszúak lehetnek, vagyis mindkét párosításból ugyanannyi él szerepel bennük. A feltétel szerint egyik út sem lehet E'-javító, M maximális volta miatt pedig egyik sem lehet M-javító. Ezért G'-ben minden út páros hosszúságú, és mindkét párosításból ugyanannyi élet tartalmaz. Ebből következik, hogy $|E'| = |M|$, vagyis az E' párosítás is maximális. □

Már csak az maradt hátra, hogy megmutassuk, miként lehet egy gráf egy adott párosításához javító utat találni. Vegyük észre, hogy az előző tétel nem csak páros gráfokról szólt. A javító út keresésénél azonban már lényegesen kihasználjuk G páros voltát. Egy $G = (V_1, V_2; E)$ páros gráfban meg tudjuk adni a csúcsok egy olyan halmazát, melyben a lehetséges javító utaknak pontosan az egyik végpontja szerepel. Például V_1 ilyen lesz, hiszen egy páros gráfban minden javító út – mivel páratlan hosszú – egy V_1 és egy V_2-beli pontot köt össze.

A javító út kereséséhez egy *alternáló erdőt* növesztünk. Ennek csúcsait szintekre osztva érdemes elképzelni. A nulladik szinten a V_1-beli párosítatlan csúcsok vannak. Ezután szintről szintre növesztgethetjük a lehetséges javító utakat úgy, hogy felváltva veszünk fel párosítatlan és párosított éleket. Az egyre bővülő gráf egy erdő lesz, melynek szintjein felváltva szerepelnek V_1-beli és V_2-beli pontok. Ha valamelyik páratlan szinten megjelenik egy még párosítatlan v csúcs, akkor javító utat találtunk. Az erdőben a v-től a nulladik szintre visszavivő út javító út lesz.

Javító út keresése alternáló erdő építésével

Adott a $G = (V_1, V_2; E)$ páros gráf, és egy E' párosítása. Egy az E'-hez tartozó alternáló erdőt a következő módon, szintről szintre haladva építhetünk fel:

0. szint:	V_1 azon pontjai, melyeket E' nem fed le, vagyis a párosítatlan pontok.
⋮	
$2k-1$. szint:	V_2 azon még fel nem vett pontjai, melyek egy párosítatlan, azaz egy $E \setminus E'$-beli éllel elérhetők egy $2k-2$. szintbeli pontból; ezen éllel együtt.
$2k$. szint:	V_1 azon még fel nem vett pontjai, melyek egy párosított, azaz egy E'-beli éllel elérhetők egy $2k-1$. szintbeli pontból; ezen éllel együtt.
⋮	

Egy csúcsot legfeljebb egyszer veszünk fel, mégpedig az első lehetséges szinten. A kialakuló gráfban nincs kör, hiszen egy csúcshoz az alacsonyabb sorszámú szintekről csak egy él illeszkedik. A fa növesztése könnyen megvalósítható a szélességi bejárás kis módosításával. Induláskor a nulladik szint pontjait tesszük a Q sorba. Ezután pedig a bejárás során a páros szintről induló élek közül csak a párosítatlanokkal foglalkozunk; a páratlan szintről indulók közül pedig csak az E'-beliekkel. Ha tehát G éllistával adott, akkor az alternáló erdőt $O(e)$ költséggel megkaphatjuk.

Állítás: *A $G = (V_1, V_2; E)$ páros gráfban akkor és csak akkor van az E' párosításra nézve javító út, ha az E'-hez tartozó alternáló erdőben valamelyik páratlan szinten megjelenik egy párosítatlan pont.*

Bizonyítás: Az elégségesség nyilvánvaló, ugyanis egy páratlan szinten levő párosítatlan pontot az erdőben a nulladik szinttel összekötő út egy javító út.

A fordított állításhoz először megmutatjuk, hogy a V_1 párosítatlan csúcsaiból (ezek vannak az erdőben a nulladik szinten) alternáló úton elérhető pontok mindegyikét beválasztjuk valamikor az alternáló erdőbe. Ennek igazolására tegyük fel, hogy $v_0, v_2, \ldots, v_k$ egy alternáló út, és $v_0 \in V_1$ egy párosítatlan csúcs; i szerinti indukcióval megmutatjuk, hogy v_i bekerül az erdőbe. Ez nyilván igaz, ha $i = 0$.

Az út v_{2j} csúcsa V_1-ben van és a (v_{2j}, v_{2j+1}) él párosítatlan. Tehát ha v_{2j}-t beválasztottuk, akkor v_{2j+1} is bekerül, ha előbb nem, akkor a v_{2j} utáni szintre. Hasonló okfejtést alkalmazhatunk a páros indexű csúcsokra: az út v_{2j-1} csúcsa V_2-ben van és $(v_{2j-1}, v_{2j}) \in E'$. Így v_{2j-1} után v_{2j} is sorra kerül, ha korábban ez még nem történt meg.

Tegyük fel mármost, hogy G-ben a v és w csúcsok egy javító út végpontjai. Ezek a pontok nyilván párosítatlanok, és egyikük – mondjuk v – a V_1-ben van. Ekkor pedig $w \in V_2$. A v szerepel az erdő nulladik szintjén, és w-t is be fogjuk

választani valamikor, hiszen v-ből alternáló úton elérhető. A w az erdőnek csak egy páratlan sorszámú szintjére kerülhet, mivel V_2-beli csúcsokat csak páratlan szintekre veszünk fel.

□

Amint azt már megállapítottuk, az alternáló erdő növesztésének költsége $O(e)$. Nyilvánvaló másfelől, hogy legfeljebb $n/2$-szer kell javító utat keresnünk. *Ezért a magyar módszer időigénye éllistával adott bemenet esetén* $O(ne)$. A témától búcsúzóban megjegyezzük, hogy vannak a magyar módszernél hatékonyabb (és persze sokkal kevésbé egyszerű) algoritmusok is. Ilyen például J. E. Hopcroft és R. M. Karp 1973-ból való módszere, ami $O(\sqrt{n}e)$ lépésben talál maximális párosítást páros gráfokban.

6.8. Maximális folyamok hálózatokban

Hempergő jóság vagy,
termést bőséggel adj...
RADNÓTI MIKLÓS: Himnusz a Nílushoz

Az emberiség történelme a kezdetektől fogva szorosan kötődik a nagy folyamokhoz. Valami hasonlót mondhatunk a kombinatorikus optimalizálás és a hálózati folyamok viszonyáról. A hálózati folyamok természetes és meglepően gazdag modellt kínálnak olyan helyezetek leírására, amelyek lényege, hogy egy rendszerben anyag áramlik bizonyos – forrásnak nevezett – keletkezési helyektől bizonyos – nyelőnek nevezett – felhasználási helyek felé. Az *anyag* itt ezerféle dolgot jelenthet; gondolhatunk csővezetéken áramló folyadékra, elektromos áramra, számítógépes hálózaton közvetített információra, utcákon mozgó járművekre stb.

A hálózat egyes csomópontjai között közvetlen összeköttetések vannak; ezeket az anyag szállítására alkalmas médiumoknak (csővezeték, kábel, utca, stb.) tekintjük. A legegyszerűbb modellben egy ilyen kapcsolatot az *irányával* és a *kapacitásával* jellemzünk. Az előbbi mondja meg, hogy a két csomópont között milyen irányú áramlás lehetséges, az utóbbi pedig az időegység alatt átvihető maximális mennyiséget jelenti. Például egy kommunikációs vonal esetén a kapacitás lehet a másodpercenként átvihető bitek száma. Az anyagáramlást, a *folyamot* úgy képzeljük el, hogy a forrásokban állandó sebességgel keletkezik az anyag, és ugyanekkora sebességgel enyészik el a nyelőkben. A többi – a forrásoktól és a nyelőktől különböző – csomóponton csak átáramlik; ami beérkezik, az távozik is. Ezeknél a csúcsoknál a folyamra teljesül az elektrodinamikából ismerős Kirchoff csomóponti törvény megfelelője. Nézzük mindezeket pontosabban, azzal az egyszerűsítő

feltevéssel, hogy a hálózatban csak egy forrás és egy nyelő van! A forrást és a nyelőt hagyományosan s (source) és t (target, terminal) jelöli.

Definíció (hálózat): *Adott egy $G = (V, E)$ irányított gráf és ennek két különböző pontja, s és t, melyeket* forrás*nak, illetve* nyelő*nek hívunk. Adott még egy az éleken értelmezett $c : E \to \mathbb{R}^+$ pozitív értékű kapacitásfüggvény. A kényelem kedvéért terjesszük ki c értelmezési tartományát: legyen $c(u, v) := 0$ minden $u, v \in V$, $(u, v) \notin E$ esetén. Ekkor a (G, s, t, c) négyest* hálózat*nak nevezzük.*

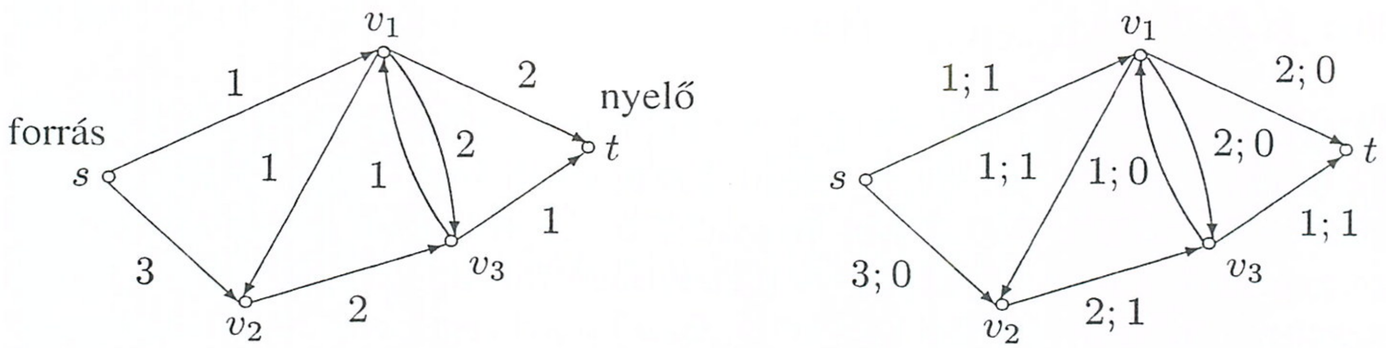
Az anyag áramlását – a folyamot – úgy írhatjuk le, hogy a hálózat minden u, v csúcspárjára megmondjuk, mennyi az időegység alatt az (u, v) összeköttetésen (vagy összeköttetéseken, ha több közvetlen kapcsolat is van u és v között) átmenő $f(u, v)$ mennyiség.

Definíció (folyam): *Az $f : V^2 \to \mathbb{R}$ függvényt* folyam*nak hívjuk, ha teljesülnek a következő feltételek:*
(1) minden u, v pontpárra $f(u, v) = -f(v, u)$,
(2) minden $u \notin \{s, t\}$ pontra $\sum_{v \in V} f(u, v) = 0$,
(3) minden u, v pontpárra $f(u, v) \leq c(u, v)$.
Ha $f(u, v) = c(u, v)$, akkor az (u, v) párat telített*nek mondjuk (akár éle G-nek, akár nem). Az f* folyam értéke*, melyet $|f|$-fel jelölünk, az s-ből kimenő összes folyam, azaz $|f| := \sum_{v \in V} f(s, v)$.*

Az (1) feltétel egy kényelmes és eléggé szemléletes technikai kikötés. Azt a tényt, hogy u-ból v-be $f(u, v)$ mennyiséget juttatunk el, úgy is nézhetjük, hogy v-ből u-ba vittünk $-f(u, v)$-t. Az $f(u, v) > 0$ eseménynek pedig azt a jelentést tulajdoníthatjuk, hogy (u, v) mentén $f(u, v)$ mennyiségű anyag megy *ki* u-ból, míg $f(u, v) < 0$ azt jelenti, hogy $-f(u, v)$ megy *be* u-ba. Erre is tekintettel a (2) követelmény a Kirchoff-törvény megfogalmazása. A (3) egyenlőtlenség fejezi ki, hogy az u-ból v be (közvetlenül, kerülő utak nélkül) átvitt mennyiség nem haladhatja meg az (u, v) kapcsolat kapacitását. Az (1) és (3) alapján világos, hogy ha (u, v) és (v, u) egyike sem éle a hálózatnak, akkor $f(u, v) = f(v, u) = 0$, vagyis az ilyen párok között nincs érdemleges történés. Ebből és az (1) feltételből következik, hogy az f folyam megadásához elegendő azon $f(u, v)$ értékek ismerete, melyekre $u \to v$ éle G-nek.

A következő rajz bal oldalán egy hálózat látható. Csak a pozitív kapacitású éleket tüntettük fel, a kapacitásukkal együtt. A jobboldali rajzon egy f folyam jellemző adatai is szerepelnek. Az éleken levő számok közül az első a kapacitás, a második a rajta átáramló mennyiség. A $v_1 \to t$ él kapacitása 2, a rajta átfolyó mennyiség pedig $f(v_1, t) = 0$. A folyam értéke $|f| = 1$, és például (v_1, v_2), valamint (t, v_1) telített párok. A v_1, v_3 csúcspár között két kapcsolat van, az egyik

$v_1 \to v_3$, a másik $v_3 \to v_1$. Ezeken most 0 folyam megy át. Az összképet illetően ugyanez lenne a helyzet, ha mindkét élen 1 egység áramlana keresztül. Ekkor is igaz lenne, hogy $f(v_1, v_3) = f(v_3, v_1) = 0$. Azt szeretnénk hangsúlyozni, hogy $f(u, v)$ meghatározásához *mindegyik* u és v között menő élet figyelembe kell venni.



A maximális folyam problémája alapvető jelentőségű algoritmikus feladat mind az elméleti, mind a gyakorlati alkalmazások szempontjából. A probléma megfogalmazása és az első megoldás is L. R. Ford és D. R. Fulkerson (1957) nevéhez fűződik.

A probléma:

Adott egy (G, s, t, c) hálózat; találjunk hozzá egy maximális értékű f folyamot.

A kérdés tehát az, hogy a hálózatot folyamatosan működtetve a forráspontból maximálisan mennyi anyag tud elindulni, illetve a nyelőpontba beérkezni egységnyi idő alatt úgy, hogy a Kirchoff-törvényt és a kapacitások adta korlátokat megtartsuk. Természetesen a válaszunkban meg kell mondanunk a szállítás útvonalait is. Előre bocsátjuk, hogy a maximális folyamok értéke szükségképpen nemnegatív. Minden hálózaton van ugyanis egy nulla értékű folyam, a *nulla-folyam*, amit az $f(u, v) := 0$ egyenlőségekkel definiálhatunk $(u, v \in V)$.

Ahhoz, hogy a megoldás közelébe jussunk, vagy egyáltalán, egy maximális folyam létezését igazolni tudjuk, érdemes a *vágás* fogalmát bevezetnünk. A következő szakaszban a kombinatorikában oly gyakran előforduló minimax tételek[9] egyikét bizonyítjuk. A tétel segítségével – amely folyamok és vágások között létesít kapcsolatot – lerakjuk a megoldás alapköveit.

[9]Egy minimax tétel egy mennyiség maximumának és egy másik mennyiség minimumának az egyenlőségét mondja ki.

6.8.1. Kapcsolat a minimális vágással: a Ford–Fulkerson-tétel

Definíció (s,t-vágás): *Legyen (G,s,t,c) egy hálózat; $A,B \subseteq V$, $A \cup B = V$ és $A \cap B = \emptyset$ (azaz A és B partícionálják V-t). Legyen továbbá $s \in A$, $t \in B$. Ekkor az A,B halmazpárt s,t-vágásnak, vagy röviden vágásnak hívjuk. Az A,B vágás* kapacitásán *a $c(A,B) := \sum_{u \in A, v \in B} c(u,v)$ mennyiséget értjük. Ha f egy folyam, akkor definiáljuk az A,B vágáson áthaladó folyamot. Ezt $f(A,B)$-vel jelölve: $f(A,B) := \sum_{u \in A, v \in B} f(u,v)$.*

Figyeljük meg, hogy $|f| = f(\{s\}, V \setminus \{s\})$. Tehát a folyam értéke az s-et valóban elhagyó folyam mennyiségét jelenti. Ezt úgy kell érteni, hogy az s-et elhagyó összes folyamból levonjuk az s-be visszatérő összes folyamot, hiszen azon v pontokra, melyekből s-be folyik pozitív mennyiségű folyam, $f(s,v)$ negatív, így az összeget a megfelelő értékkel csökkenti. Ezzel összhangban az A,B vágáson áthaladó folyam azt fejezi ki, hogy mekkora mennyiségű anyag jut át rendszerünk A „partjáról" a B-re. Nyilván ugyanannyi, mint amennyi s-ből elindul, mivel útközben nem hagyunk ott sehol semmit. Ezt fejezi ki szabatosan a következő lemma:

Lemma: *Tetszőleges A,B s,t-vágásra és f folyamra $|f| = f(A,B)$.*

Bizonyítás: Figyelembe véve, hogy $B = V \setminus A$, az $f(A,B)$ összeg így írható:

$$f(A,B) = \sum_{u \in A, v \in B} f(u,v) = \sum_{u \in A, v \in V} f(u,v) - \sum_{u \in A, v \in A} f(u,v) = |f| - 0 = |f|.$$

Itt $\sum_{u \in A, v \in V} f(u,v) = |f|$, mert ha $u \neq s$, akkor $\sum_{v \in V} f(u,v) = 0$ a folyam definíciójának (2) feltétele miatt; $u = s$ esetén pedig az összeg a folyam értékét adja. Az (1) feltétel szerint meg $f(u,v) + f(v,u) = 0$, ezért $\sum_{u \in A, v \in A} f(u,v) = 0$. □

Ezen a ponton láthatjuk, hogy a forrásnál keletkező anyag hiánytalanul eljut a nyelőhöz, ha a szállítás útját-módját a folyam-definícióban kikötöttek szerint választjuk meg. Az előző lemma alkalmazható ugyanis az $A = V \setminus \{t\}$, $B = \{t\}$ speciális esetre. Arra jutunk, hogy $|f| = f(V \setminus \{t\}, \{t\})$; a jobb oldalon éppen a t-ben elnyelődő mennyiség szerepel.

Az A,B vágás $c(A,B)$ kapacitása korlátot ad az A-ból B-be átvihető anyag mennyiségére. Szemléletesen azt várnánk, hogy $s \in A$ és $t \in B$ miatt a vágás kapacitása korlátozza az $|f|$ értéket is. Ez így is van:

Állítás: *Tetszőleges A,B s,t-vágásra és f folyamra $|f| \leq c(A,B)$.*

Bizonyítás:

$$|f| = \sum_{u \in A, v \in B} f(u,v) \leq \sum_{u \in A, v \in B} c(u,v) = c(A,B).$$

Az első egyenlőségnél a lemmát, az egyenlőtlenségnél pedig a folyam-definíció (3) feltételét használtuk. □

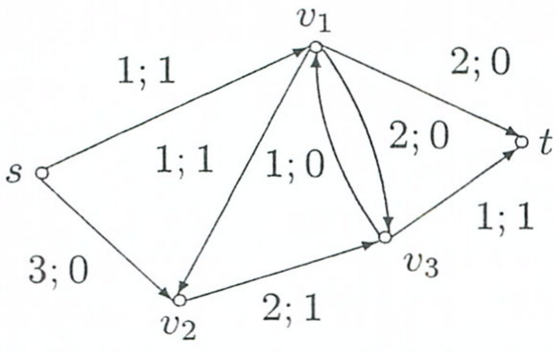
Tehát egy folyam értéke nem lehet nagyobb egy tetszőleges vágás kapacitásánál. A megígért minimax tétel így már sejthető: az egyenlőség el is érhető, azaz a maximális folyam értéke egyenlő a minimális vágáskapacitással. Hogy ezt bizonyítani tudjuk, szükségünk lesz néhány további fogalomra.

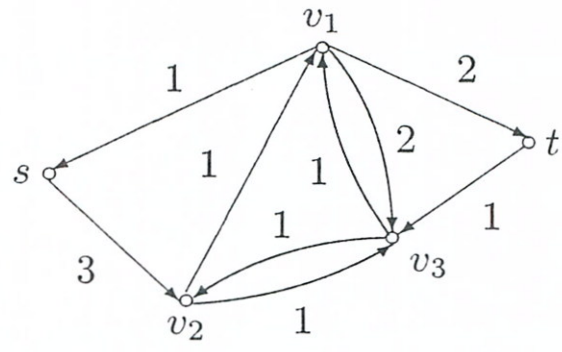
A javító gráf

Tegyük fel, hogy már eljutottunk valameddig egy nagy folyam tervezgetésében. Van egy f folyamunk, és az érdekel bennünket, hogy lehet-e javítani rajta, és ha igen, akkor hogyan. E célból nézzünk a folyam rajzára, és próbáljuk meg feltérképezni a még kiaknázatlan lehetőségeket. Minden egyes (u, v) csúcspárra vizsgáljuk meg, hogy mennyi anyagot tudnánk *még* átvinni közvetlenül u-ból v-be. A folyam-definíció (3) feltétele szerint az (u, v)-n átvihető mennyiség legfeljebb $c(u, v)$, ezért a növelési lehetőség e kapcsolat mentén $c(u, v) - f(u, v)$. Ezek a mennyiségek nem negatívak, hiszen f egy folyam; az f növelésére ott van esélyünk, ahol $c(u, v) - f(u, v) > 0$. Ezek az észrevételek alapot nyújtanak a következő fogalomhoz.

Definíció (javító gráf): *Adott egy* $(G = (V, E), s, t, c)$ *hálózat, és rajta egy* f *folyam. Jelölje* $r(u, v) := c(u, v) - f(u, v)$ *a maradék kapacitások függvényét. Az* f *folyamhoz tartozó* javító gráf *a* $G_f = (V, E_f)$ *gráf az élein értelmezett* r *kapacitásfüggvénnyel, ahol* $E_f = \{(u, v); r(u, v) > 0\}$.

A következő ábra bal felén az előző példabeli hálózat és folyam szerepel. Mellette a folyamhoz tartozó G_f javító gráf látható, az éleken a maradék kapacitásokkal. Vegyük észre, hogy G_f-nek vannak olyan élei is, amelyek G-nek nem élei. Ilyen például a $t \to v_3$. Ennek jelenléte indokolt, minthogy $c(t, v_3) = 0$, $f(t, v_3) = -1$, amiből $r(t, v_3) = 1$.





Az r függvény pozitív értékeket vesz fel a G_f élein, így (G_f, s, t, r) egy hálózat. Könnyen belátható, hogy egy $f' : V^2 \to \mathbb{R}$ függvény akkor és csak akkor folyam G_f-ben, ha $f + f'$ folyam G-ben; ekkor pedig $|f + f'| = |f| + |f'|$. Ha tehát találunk egy pozitív értékű f' folyamot G_f-ben, akkor $f + f'$ egy az f-nél nagyobb értékű folyam lesz az eredeti hálózatban. A folyam növelésének feladatát ezzel visszavezettük egy rokon kérdésre: pozitív értékű folyamot kell találnunk a (G_f, s, t, r) hálózatban.

Hogyan keressünk ilyen folyamot? Tegyük fel, hogy G_f-ben találunk egy irányított $s \rightsquigarrow t$ utat, és az úton szereplő élek kapacitásainak minimuma $d > 0$. Ez az út egy d értékű f' folyamot határoz meg G_f-ben: f' vegyen fel d-t az út élein, $-d$-t az út éleinek fordítottjain és nullát az összes többi páron.

Definíció (növelő út, kritikus él): *Legyen (G, s, t, c) hálózat, és f egy folyam rajta. A G_f-beli irányított $s \rightsquigarrow t$ utakat* növelő utak*nak hívjuk. Egy növelő úton szereplő élek maradék kapacitásainak minimumát az úthoz tartozó* kritikus kapacitás*nak, a megfelelő éleket pedig* kritikus élek*nek*[10] *nevezzük.*

Példánkban $sv_2v_3v_1t$ növelő út, ami mentén eggyel növelhetjük a folyamot. Növelő út sv_2v_1t is, ahol a (v_2, v_1) él menti növelés az eredeti gráfban a (v_1, v_2) él menti csökkentést jelenti (kevesebbet viszünk a rossz irányba). Mindkét növelő út kritikus kapacitása 1. Az első út kritikus élei $v_2 \to v_3$ és $v_3 \to v_1$, a másodiké pedig egyedül $v_2 \to v_1$.

Az eddigiekből kezd alakot ölteni egy algoritmus: a meglevő f folyamunkhoz készítsük el a G_f javító gráfot, ebben keressünk növelő utat, e mentén növeljük meg a folyam értékét a kritikus kapacitással; azután ezt ismételjük. Kiindulásnak jó lesz a nulla-folyam. Kérdés, mi van akkor, ha már nincs G_f-ben növelő út. A következő tétel egyebek között azzal biztat bennünket, hogy ebben az esetben készen vagyunk, vagyis f már maximális.

[10]A *kritikus él* kifejezést - egészen más jelentéssel - már használtuk a PERT-módszer kapcsán. A két fogalom között nincs tartalmi kapcsolat.

Tétel (Ford–Fulkerson-tétel): *Legyen f egy folyam a (G, s, t, c) hálózatban. A következő állítások egyenértékűek:*

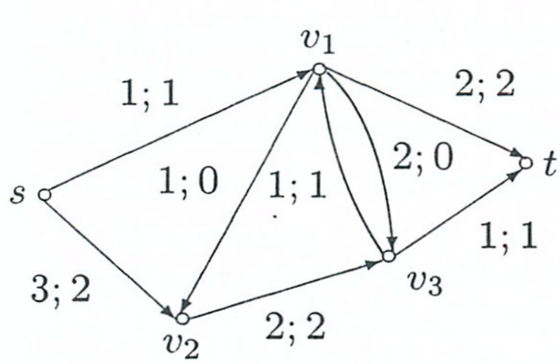
(a) f egy maximális (értékű) folyam;

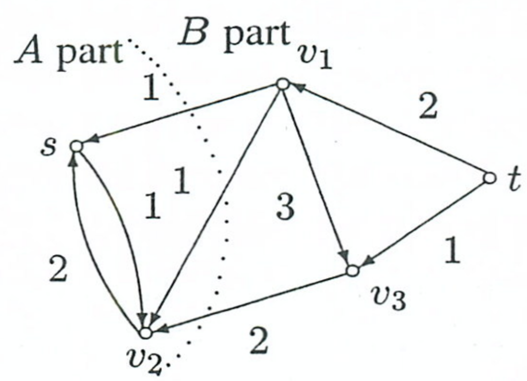
(b) f-hez nem létezik növelő út;

(c) létezik olyan A, B s, t-vágás, melyre $c(A, B) = |f|$.

Bizonyítás:
(a)⇒(b): Ha f-hez létezik növelő út a G_f gráfban, akkor f értéke az előzőek szerint ennek az útnak a $d > 0$ kritikus kapacitásával növelhető; ekkor f nem lehet maximális, mert van $|f| + d$ értékű folyam is.
(b)⇒(c): Tegyük fel, hogy nem létezik növelő út f-hez. Legyen A azon pontok halmaza, amelyek G_f-ben s-ből irányított úton elérhetők, és legyen $B := V \setminus A$. Nyilvánvaló, hogy $s \in A$, és $t \in B$. Az A, B pár tehát egy s, t-vágás. Legyenek most $u \in A$, $v \in B$ tetszőleges pontok. Az (u, v) pár nem éle G_f-nek. Ez annyit tesz, hogy az (u, v) pár telített, vagyis $c(u, v) = f(u, v)$. Ezeket az egyenlőségeket összegezve az összes $u \in A$, $v \in B$ párra azt kapjuk, hogy $c(A, B) = f(A, B)$. A lemmából tudjuk, hogy $f(A, B) = |f|$, amiből $c(A, B) = |f|$.
(c)⇒(a): Az állítás szerint $|f| \leq c(A, B)$ igaz bármely f folyamra és A, B vágásra. Az $|f| = c(A, B)$ egyenlőség ezért csak úgy lehetséges, hogy f egy maximális folyam, A, B pedig egy minimális (kapacitású) vágás. □

A következő ábra a Ford–Fulkerson-tételben felmerült helyzetet szemlélteti. A bal oldali rajzon a korábban már vizsgált hálózat egy f maximális folyamát mutatjuk. Látható, hogy $|f| = 3$. Mellette a folyamhoz tartozó javító gráf található; ezen feltüntettük a bizonyításból adódó A, B vágást. A vágás A partját az s és v_2 csúcsok alkotják. Az eredeti hálózat rajzára pillantva meggyőződhetünk róla, hogy a vágás kapacitása $c(A, B) = c(s, v_1) + c(v_2, v_3) = 1 + 2 = 3$, ami éppen a folyam értéke.





A Ford–Fulkerson-tétel érdekes következménye, hogy egy folyam maximális voltának van egyszerűen és gyorsan ellenőrizhető bizonyítványa. Képzeljük el,

hogy valaki mutat nekünk egy kusza hálózatot, rajta egy f folyammal, és azt állítja, hogy f maximális. Hogyan győzhet meg bennünket erről egyszerűen? A tétel szerint elegendő egy olyan A, B vágást mutatnia, aminek a kapacitása éppen a folyam értéke. Mivel $|f|$ és $c(A, B)$ is gyorsan kiszámolható, egykettőre bizonyosságot szerezhetünk az állítás igazságáról. Az algoritmikus problémáknak azzal a tulajdonságával, hogy a válasznak van-e hatékonyan ellenőrizhető bizonyítványa, a 8. fejezetben foglalkozunk alaposabban.

6.8.2. A Ford–Fulkerson-algoritmus

A maximális folyam keresésére szolgáló *Ford–Fulkerson-algoritmus* lényegi lépéseit tulajdonképpen már elmondtuk: folyamok $f_0, f_1, \ldots, f_k = f^*$ sorozatát konstruáljuk meg sorban egymás után. Az f_0 az azonosan nulla folyam. Az f_i birtokában f_{i+1}-et úgy kapjuk, hogy az f_i javító gráfjában keresünk egy javító utat; az út mentén a d_i kritikus kapacitással növelve kapjuk az f_{i+1} folyamot. Érvényes tehát az $|f_{i+1}| = |f_i| + d_i$ összefüggés. Akkor állunk meg, amikor a folyamhoz már nem létezik növelő út. A Ford–Fulkerson-tételből következik, hogy ekkor egy maximális folyamunk van.

Feladat: Mutassuk meg, hogy ha a hálózat éllistával adott, akkor a növelő lépés, vagyis f_i-ből f_{i+1} számítása $O(e)$ elemi művelettel kivitelezhető, ahol e a G éleinek száma. (Feltehető, hogy G összefüggő. A G_{f_i} javító gráfnak legfeljebb csak $2e$ éle lehet, így a hálózat és f_i ismeretében $O(e)$ költséggel megépíthető. Javító utat ezután a G_{f_i} gráf s-ből rajtoló szélességi bejárásával kaphatunk.)

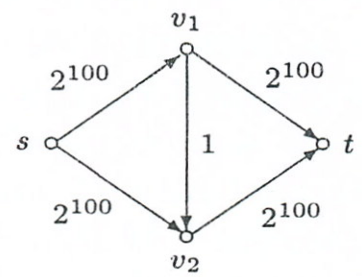
Mit mondhatunk a növelő lépések számáról? Ha az élkapacitások egészek, akkor legfeljebb $|f^*|$ számú növelés után készen vagyunk (f^* egy maximális folyamot jelöl), ugyanis ekkor a d_i kritikus kapacitások pozitív egészek lesznek. Rögzítsük ennek a fejtegetésnek egy fontos következményét:

Következmény: *Legyen (G, s, t, c) egy hálózat, és tegyük fel, hogy a $c(u, v)$ kapacitások mind egész számok. Ekkor van olyan f^* maximális folyam, hogy az $f^*(u, v)$ értékek egészek ($u, v \in V$). Ebből adódóan a maximális folyam értéke is egész.*

Bizonyítás: A Ford–Fulkerson-algoritmus működéséből látható, hogy az $f_i(u, v)$ értékek egészek lesznek. Ez nyilvánvaló a nulla-folyamra. Utána pedig használhatjuk, hogy az $f_{i+1}(u, v) - f_i(u, v)$ különbség a $0, \pm d_i$ egészek valamelyike. □

Ha tehát a kapacitások egészek, akkor legfeljebb $|f^*|$ növeléssel eljutunk egy maximális folyamig. Ez egyfelől megnyugtató, hiszen látjuk, hogy véges sok lépés

elég. Másfelől a növelések száma ettől még égbekiáltóan nagy is lehet, amint azt alábbi hálózat mutatja.



Ha ebben a hálózatban felváltva mindig az sv_1v_2t és az sv_2v_1t növelő utak mentén javítunk (a kritikus kapacitás 1 lesz), akkor a nulla-folyamból 2^{101} javítás után kapunk maximális folyamot. Ugyanakkor két javító menet is elég lenne: először az sv_1t, majd az sv_2t út mentén.

Ha az élkapacitások racionális számok, akkor a közös nevezőjükkel való szorzás után egész kapacitásaink lesznek. Így ebben az esetben is véges sok (noha esetleg nagyon sok) növelő lépést hajtunk végre.

Ha irracionális kapacitásokat is megengedünk, akkor előfordulhat, hogy a javító utak balszerencsés választása miatt az eljárás nem ér véget véges sok lépésben. Sőt, még az is megeshet, hogy az $|f_i|$ értékek a maximális folyamokénál kisebb értékhez konvergálnak.

Mindezekből nyilvánvaló, hogy nagyon nem mindegy, miként választunk növelő utat. A találomra való választással racionális kapacitások esetén ugyan egy működő, ámde lassú algoritmust kapunk, irracionális kapacitásokkal pedig a módszer nem is működik.

6.8.3. Edmonds–Karp és Dinic algoritmusai

J. Edmonds és R. M. Karp (1972) nagyon egyszerűen hangzó receptet javasolt az előző szakaszban említett bajok orvoslására. Módszerük rokonszenves vonása, hogy a javító menetek számára csak n-től és e-től (azaz csupán a hálózat pontjainak és éleinek számától) függő korlát adható. Javaslatuk a Ford–Fulkerson-módszer pontosítása, amennyiben arról rendelkezik, hogy miként válasszunk növelő utat az adott f folyamhoz:

> EDMONDS–KARP-HEURISZTIKA: A folyam növelésére mindig a legrövidebb – vagyis a legkevesebb élből álló – növelő utak egyikét válasszuk.

Edmonds és Karp ötletéről nem látszik elsőre, hogy miért olyan jó. Az mindenesetre biztató, hogy az előző rajz hálózatában a nulla-folyamból kiindulva két menetben eljut a maximálisig.

A gondosabb elemzéshez tegyük fel, hogy adott a (G, s, t, c) hálózat és rajta egy f folyam. Tekintsük a G_f javító gráfot; legyen benne π egy legrövidebb növelő út. Ennek hosszát (élszámát) jelölje l. Osszuk rétegekre a G_f csúcsait az s-től való távolság szerint. Az s-ből egyetlen irányított éllel elérhető csúcsok alkotják az első réteget, és t az l-edik rétegbe kerül. Ezt a beosztást kapjuk, ha s-től indulva szélességi bejárással végigmegyünk az s-ből irányított úton elérhető v csúcsokon, és közben számítjuk a $D[v]$ értékeket (lásd a 6.5. szakaszt). Világos, hogy a v és w csúcsok pont akkor vannak egy rétegben, ha $D[v] = D[w]$. Kiemeljük a felosztás két fontos tulajdonságát:

(1) Egy él legfeljebb egy réteggel mehet előre.

Ez nyilvánvaló, hiszen az s-től i távolságra levő pontból kiinduló él végpontjának a távolsága nem lehet több, mint $i + 1$. A G_f egy $x \longrightarrow y$ élét nevezzük *vastagnak*, ha $D[y] = D[x] + 1$. Az (1) tény hasznos folyománya a következő:

(2) Az l hosszúságú $s \leadsto t$ utak csupa vastag élből állnak, és nincs l-nél rövidebb $s \leadsto t$ út.

Nézzük most, miként módosul a kép, ha π mentén növeljük f-et a kritikus kapacitással. Hogyan kapható meg az új f' folyam $G_{f'}$ javító gráfja G_f-ből? A π út legalább egy éle – nevezetesen egy kritikus él – telített lesz, így eltűnik a javító gráfból. Ezenkívül legfeljebb l darab új él jelenik meg a $G_{f'}$-ben (π élei ellenkező irányítással, ha eddig még nem szerepeltek), de ezek az új élek mind visszafelé, alacsonyabb sorszámú rétegbe mennek. Arra jutunk innen, hogy (1) és (2) a $G_{f'}$-re vonatkozóan is igaz marad (noha esetleg a rétegezés már nem tükrözi hűen a távolságokat). Ebből viszont azonnal látszik, hogy *a következő növelő út sem lehet rövidebb l-nél.*

Nézzük meg, hogy hányszor adódhat egymás után l hosszú növelő út. Ha van ilyen $G_{f'}$-ben, akkor a növelés utáni javító gráfban eggyel kevesebb vastag él lesz (legalább egy kritikus él törlődik), és az (1), (2) tények is érvényben maradnak, mert a növelés eredményeként csak visszafelé mutató élek keletkezhetnek.

A (2) tulajdonság öröklődése miatt addig lesz l élből álló növelő út, amíg marad vastag élekből álló $s \leadsto t$ út. A vastag élek viszont minden növelésnél fogynak. Így legfeljebb e ilyen növelés lehetséges. Érvényes tehát a következő:

Tétel: *Az Edmonds–Karp-heurisztika szerinti növelésnél a növelő utak hosszai nem csökkenő sorozatot alkotnak. Ebben a sorozatban egy adott úthosszúság legfeljebb e-szer fordulhat elő. Következésképpen legfeljebb $e(n-1)$ növelés lehetséges. A heurisztika alkalmazásával $O(e^2 n)$ elemi lépésben kapunk maximális folyamot.*

Bizonyítás: Az első két állítást már beláttuk. A harmadik – a növelések számáról szóló – azonnal következik ezekből, hiszen a növelő utak egyszerű utak. A lehet-

séges hosszúságaik ezért $1, 2, \ldots, n-2, n-1$, és bármely hosszúság legfeljebb e-szer léphet fel.

Az utolsó állításhoz elég igazolni, hogy *egy* növelés $O(e)$ időben elvégezhető. A növelés költségéről korábban mondottakhoz csak annyit kell hozzátenni, hogy a javító gráfban az s-ből indított szélességi bejárásnál a faéleket követve éppen egy legrövidebb $s \rightsquigarrow t$ utat kapunk. □

A bizonyítást jobban szemügyre véve kiderül, hogy f javításakor az l hosszú növelő utak élei mind vastag élek G_f-ben. Az összes ilyen javítás műveletigényére $O(e^2)$ korlátot kaptunk (legfeljebb e növelés darabonként $O(e)$ költséggel). Kihasználva, hogy a vastag élekből álló gráf egy DAG, ezek a növelések gyorsabban is elvégezhetők. De nézzük pontosabban, mi is a cél:

Legyen (H, s, t, r) éllistával adott hálózat, ahol H egy DAG, és r az élein értelmezett kapacitásfüggvény. Olyan g folyamot – ún. blokkoló folyam*ot keresünk, amelyhez minden $s \rightsquigarrow t$ irányított úton van telített él.*

Feladat: Igazoljuk, hogy egy maximális folyam egyben blokkoló folyam is. Mutassuk meg, hogy a megfordítás nem igaz: adjunk példát blokkoló folyamra, ami nem maximális.

Ha találunk egy g blokkoló folyamot a G_f vastag éleiből álló H gráfban, akkor $f + g$ már biztosan nem növelhető a vastag élek mentén, azaz minden további növelés csak l-nél hosszabb úton lehetséges.

Dinic módszere DAG-beli blokkoló folyam keresésére

Tegyük fel, hogy a H DAG-nak n csúcsa és m éle van. E. A. Dinic (1970) algoritmusa a nulla-folyamból indulva szerkeszt egy g blokkoló folyamot. Az algoritmus történései három csoportba sorolhatók; ezek a *nyomulás*, a *növelés* és a *takarítás*.

Nyomulás során egy s-ből induló π irányított utat építünk. Amíg csak lehetséges, π-t az utolsó pontjából induló valamelyik él végpontjával bővítjük. Akkor állunk meg a nyomulással, amikor vagy t-be értünk, vagy már nem tudunk tovább menni, mert nem vezet ki él a π utolsó pontjából. Mivel H egy DAG, a π sosem állhat több, mint n csúcsból. Ebből következően egy nyomulás $O(n)$ költséggel megvalósítható.

Növelésre akkor kerül sor, ha a nyomulással t-be értünk. Ez esetben π egy növelő út. Az aktuális g folyamot π mentén növeljük a d_π kritikus kapacitással. Ezután az út éleinek kapacitását d_π-vel csökkentjük, és töröljük a kritikus éleket (amelyek kapacitása nullára változott). Egy növelés műveletigénye arányos a π hosszával, vagyis $O(n)$-nel becsülhető.

Ha a nyomulással egy $x \neq t$ csúcsnál akadunk el (azaz nem tudunk tovább menni), akkor *takarítás* következik. Ez azt jelenti, hogy töröljük H-ból az x-be menő éleket. Egy takarítás a törölt élek számával arányos lépésszámban véghezvihető.

A takarítás és a növelés végeztével (s-ből induló) nyomulásba kezdünk. A munka akkor fejeződik be, amikor már nincs s-ből kimenő él.

Tétel: *Dinic módszerével* [11] $O(mn)$ *művelettel kapunk blokkoló folyamot a* H *DAG-ban.*

Bizonyítás: A módszer helyessége nyilvánvaló: amikor ugyanis egy élet törlünk, akkor rajta keresztül már nem vezethet az aktuális g-t növelő út. Így a munka végeztével nem létezik csupa H-beli élekből álló növelő út. Úgy is fogalmazhatunk, hogy minden $s \rightsquigarrow t$ irányított úton van telített él, tehát a végső g egy blokkoló folyam H-ban.

Ami a költségeket illeti, vegyük észre, hogy a takarítások és a növelések száma együttesen sem lehet több m-nél, mivel ezek élek törlésével járnak. A takarítások összköltsége $O(m)$, ugyanis ezek során legfeljebb m élet törölhetünk; a növeléseké pedig $O(mn)$. Nézzük most a nyomulásokat: a legutolsó kivételével minden nyomulást növelés vagy takarítás követ, így legfeljebb $m + 1$ nyomulás lehetséges. Ezek együttes lépésszáma ezért $O(mn)$. Mindent számba véve $O(m) + O(mn) + O(mn) = O(mn)$ lépés elegendő. □

Ha az Edmonds–Karp-heurisztika helyett Dinic módszerét alkalmazzuk az f folyam növelésére, akkor legfeljebb $n - 1$ ilyen növelésre lesz szükség. Az előző tétel alapján egy növelés költsége $O(en)$, mert a vastag élekből álló gráfnak legfeljebb e éle lehet. A Dinic-algoritmussal tehát $O(en^2)$ *lépésben kapunk maximális folyamot.*

Megjegyezzük, hogy DAG-beli blokkoló folyam keresésére van $O(n^2)$ lépésszámú módszer is, amivel $O(n^3)$ költséggel adódik maximális folyam. Az első ilyen módszert A. V. Karzanov találta 1974-ben. A folyamalgoritmusoktól búcsúzóban megemlítjük, hogy a legjobb ismert módszerek költsége $O(en\log(n^2/e))$ (A. V. Goldberg, R. E. Tarjan, 1986) és $O(en)$, ha $e > n^{5/3+\epsilon}$ (N. Alon, J. Cheriyan, T. Hagerup, 1990). Érdekes nyitott kérdés, hogy létezik-e minden hálózaton $O(en)$ lépésszámban célt érő folyamalgoritmus.

[11] Valójában Dinic módszerének egyszerűsített változatát ismertettük. Például takarítás után nem kell feltétlenül s-től kezdeni a nyomulást. Jobban járunk, ha csak a π út x előtti pontjáig megyünk vissza, és onnan nyomulunk előre.

6.8.4. Alkalmazások

Három példával szeretnénk érzékeltetni a folyamok széleskörű és sokszínű alkalmazási lehetőségeit. Hogy egyszerűsítsünk a dolgokon, olyan hálózatokra szorítkozunk, amelyekben bármely u, v pontpár között legfeljebb csak az egyik irányba megy él. Ezáltal a $v \rightarrow w$ élen átfolyó mennyiséget azonosíthatjuk az $f(v, w)$ értékkel. Így persze az is feltehető, hogy ha $v \rightarrow w$ éle a hálózatnak, akkor $f(v, w) \geq 0$. Amikor pedig egy g folyamot adunk meg, akkor elég a $g(v, w)$ értékeket leírni, ahol $v \rightarrow w \in E$.

Példáink kapcsán az olvasó könnyen meggyőződhet róla, hogy a javasolt egyszerűsítés nem jelenti az általánosság csorbítását. Ha ugyanis $u \rightarrow v$ és $v \rightarrow u$ is éle a hálózatnak, akkor utóbbit helyettesíthetjük egy $v \rightarrow v^* \rightarrow u$ élpárral, ahol v^* egy új csúcs. A $v \rightarrow v^*$ és a $v^* \rightarrow u$ élekre ugyanazt (pl. kapacitást) írjuk, mint ami a $v \rightarrow u$ élen szerepel.

Élidegen utak irányított gráfokban

Legyen G egy irányított gráf, s és t két csúcsa. *Szeretnénk minél több G-beli $s \rightsquigarrow t$ irányított utat találni azzal a feltétellel, hogy bármely két út élidegen (másként mondva: közös él nélküli) legyen.*

Nézzük evégből a (G, s, t, c) hálózatot, ahol a G minden (u, v) élének a $c(u, v)$ kapacitása 1. A kapacitások egészek, tehát van olyan f maximális folyam, amelyre az $f(u, v)$ értékek egészek minden $u, v \in V$ csúcspárra. Ebből azonnal következik, hogy ha $u \rightarrow v \in E$, akkor $f(u, v)$ vagy 0 vagy 1. Tegyük fel, hogy a folyam értéke k. Megmutatjuk, hogy ekkor van G-ben k darab páronként élidegen $s \rightsquigarrow t$ út. Ennek első lépéseként ajánljuk a következő feladatot:

Feladat: Mutassuk meg, hogy ha $k > 0$, akkor van olyan π út s-ből t-be, amelynek az élein f 1-et vesz fel. (Induljunk el s-ből egy olyan $u \rightarrow v \in E$ élen, amelyre $f(u, v) = 1$. A v-be érve töröljük ezt az élet, majd lépjünk tovább egy olyan $v \rightarrow w$ élen, melyre $f(v, w) = 1$, és töröljük ezt az élet is. Így folytatva sosem akadhatunk el egy $x \neq t$ csúcsban; x-ből ugyanis legalább annyi 1-es indul ki, mint amennyi befut oda. Előbb-utóbb t-be érünk, mert az élek fogynak.)

Ha tehát $k > 0$, akkor nézzünk egy a feladat szerinti π utat. Töröljük G-ből a π éleit, és feledkezzünk meg a rajtuk átmenő egységnyi folyamról. A kapott gráfban egy $k - 1$ értékű folyam marad, amire $k - 1 > 0$ esetén ismételjük az előző útkeresést. Így folytatva k darab élidegen irányított $s \rightsquigarrow t$ utat kapunk.

Fordítva, ha $\pi_1, \pi_2, \ldots, \pi_k$ páronként élidegen irányított $s \rightsquigarrow t$ utak G-ben, akkor ezek mindegyikén egységnyi folyamot küldhetünk a forrásból a nyelőbe,

ami összesen egy k értékű folyamot jelent. A maximális folyam értéke tehát megegyezik a legnagyobb élidegen $s \rightsquigarrow t$ utakból álló rendszer elemszámával.

Az eddigiek egy $O(en^2)$ költségű algoritmust sugallnak egy ilyen útrendszer megtalálására: először kiszámítunk egy f egész értékkészletű maximális folyamot, majd ebből összerakunk egy maximális útrendszert. A költségek közül a domináns rész a Dinic-módszer lépésszáma, amivel az f folyamot számítjuk.

Feladat: Mutassuk meg, hogy f ismeretében $O(e)$ lépésben kaphatunk k darab páronként élidegen $s \rightsquigarrow t$ utat. (Lásd az előző feladathoz mellékelt ötletet.)

Mielőtt továbbmennénk, lássuk, mit mond esetünkben a Ford–Fulkerson-tétel. A tétel szerint van egy A, B vágás, aminek a $c(A, B)$ kapacitása éppen az útjaink k száma. Más szóval A-ból B-be k irányított él megy. Ez a k él *lefogja* az összes $s \rightsquigarrow t$ utat abban az értelemben, hogy minden ilyen út tartalmaz A-ból B-be menő élet (mert $s \in A$ és $t \in B$). Ezzel egyrészt egy másik bizonyítást kaptunk arra, hogy k-nál több páronként élidegen út nem létezhet. Másfelől lényegében igazoltuk K. Menger nevezetes gráfelméleti tételét, amely szerint *a G-beli páronként élidegen $s \rightsquigarrow t$ utak maximális száma megegyezik az $s \rightsquigarrow t$ utakat lefogó élek minimális számával.*

Itt $maximum \leq minimum$ nyilvánvaló, mert egy lefogó élrendszer az útjaink mindegyikéből tartalmaz élet. Másfelől az előbb kiderült, hogy van olyan páronként élidegen $s \rightsquigarrow t$ utakból álló rendszer, ami éppen annyi útból áll, mint egy lefogó élhalmaz (utóbbi az A-ból B-be menő élek összessége). Innen pedig már adódik a $maximum \geq minimum$ egyenlőtlenség.

Maximális párosítás páros gráfokban

> *Egyszerű megfogalmazása ellenére ... máig is talán ez az egész párosításelmélet kulcseredménye.*
> LOVÁSZ LÁSZLÓ, MICHAEL D. PLUMMER[12]

Egy olyan problémát fogalmazunk meg a folyamok nyelvén, amivel már találkoztunk: a maximális párosítás keresését páros gráfokban. Ezt elsősorban a megfeleltetés egyszerűsége és elméleti érdekessége miatt tesszük, és nem azért, mert gyors algoritmust remélünk belőle. A folyamalgoritmusokra épülő megoldások nem tudnak versenyezni a párosítási feladat sajátosságaira kihegyezett közvetlen módszerekkel, így a magyar módszerrel sem.

[12]Ezzel a méltatással vezetik fel König Dénes minimax tételét a párosítások elméletével foglalkozó nagymonográfiájuk (*Matching Theory*, Akadémiai Kiadó, Budapest, 1986) 4. oldalán.

Legyen $G = (V_1, V_2; E)$ egy páros gráf, amiben maximális párosítást keresünk. Készítsünk belőle hálózatot az alábbiak szerint: először is irányítsuk az E-beli éleket V_1-től V_2 felé; ezek kapacitása legyen $|V_1| + 1$. Vegyünk fel még a meglevő pontokon kívül egy s forrást és egy t nyelőt. Az s-ből vezessünk 1 kapacitású éleket a V_1 pontjaiba, és indítsunk ugyancsak 1 kapacitású éleket a V_2 pontjaiból t-be.

Megmutatjuk, hogy *az így kapott hálózatban a maximális folyamok értéke megegyezik a G-beli maximális párosítások méretével.* Legyen f egy csupa egész értékű maximális folyam, és legyen $|f| = k$. A V_1 pontjaiba legfeljebb egységnyi folyam folyhat be, ezért egy $u \in V_1$ ponthoz legfeljebb egy olyan $(u, v) \in E$ él illeszkedhet, amelyre $f(u, v) = 1$. A többi ilyen élen f nullát vesz fel. Hasonló érvényes a V_2 pontjaira. Ezek szerint azok az $(u, v) \in E$ élek, melyekre $f(u, v) = 1$, egy párosítást alkotnak G-ben. A rajtuk átmenő összes folyam egyrészt megegyezik az élek számával. Másrészt a lemma szerint k-val is, hiszen ez a mennyiség az $\{s\} \cup V_1, \{t\} \cup V_2$ vágáson átáramló $f(\{s\} \cup V_1, \{t\} \cup V_2)$ folyam. Van tehát a G-ben k élből álló párosítás.

Fordítva, ha az $(u_1, v_1), (u_2, v_2), \ldots, (u_k, v_k) \in E$ élek egy párosítást alkotnak ($u_i \in V_1, v_i \in V_2$), akkor az $s \to u_i \to v_i \to t$ utak mindegyikén egységnyi folyamot vihetünk a hálózatunkban s-ből t-be; ezek összesen egy k értékű folyamot adnak. A maximális folyamok értéke tényleg egyenlő a maximális párosítások élszámával.

Próbáljuk meg itt is értelmezni a Ford–Fulkerson-tétel állítását, ahogy azt az élidegen utaknál tettük! A tétel alapján létezik egy A, B s, t-vágás, amelynek a k kapacitása éppen a maximális folyamok értéke, vagyis a maximális párosítások élszáma. Legyenek $x_1, x_2, \ldots, x_l$ a V_1 azon pontjai, amelyek *nincsenek* A-ban, és legyenek $y_1, y_2, \ldots, y_m$ az A halmaz V_2-beli pontjai. Az A-ból B-be menő élek között nem lehet olyan, ami V_1-ből V_2-be mutat, mert egy ilyen él kapacitása túl nagy: $|V_1| + 1 > k$. Ebből arra következtethetünk, hogy az E-beli élek mindegyike illeszkedik az x_i vagy az y_j pontok valamelyikéhez. Az x_i és az y_j pontok ebben az értelemben egy a G éleit *lefogó ponthalmazt* alkotnak. Mekkora ez a halmaz? Az A-ból B-be menő élek éppen az $s \to x_i$ és az $y_j \to t$ élek. Ezek kapacitása 1, ami azt jelenti, hogy a számuk éppen a vágás kapacitása, k. Van tehát a G-ben a maximális párosítások élszámával egyező méretű lefogó ponthalmaz.

Ezzel a birtokunkba jutott a gráfelmélet egyik tündöklő ékköve, König Dénes minimax tétele, amely így hangzik: *egy G páros gráf maximális párosításainak a mérete megegyezik a G éleit lefogó ponthalmazok minimális elemszámával.*

A $maximum \leq minimum$ egyenlőtlenség itt is nyilvánvaló: egy k élű párosítás éleit nem lehet k-nál kevesebb ponttal lefogni. Az előző okfejtés adja a jóval tartalmasabb $maximum \geq minimum$ egyenlőtlenséget.

Feladat: A bevezető fejezetben a lovagok és udvarhölgyek problémájáról megemlítettük, hogy pontosan akkor *nincs* megoldása, ha a megfelelő gráfban van König-akadály. Bizonyítsuk be ezt az állítást. (Alkalmazzuk König minimax tételét.)

Hálózatok alsó korlátokkal

A hálózatfogalom, amivel eleddig foglalkoztunk, meglehetősen egyszerűnek mondható. Gyakran előfordul, hogy az alkalmazások követelményei ennél finomabb, több tényezőre is érzékeny modelleket kívánnak meg. Most – rövid ízelítő gyanánt – egy ilyen árnyaltabb hálózat folyamairól ejtünk pár szót.

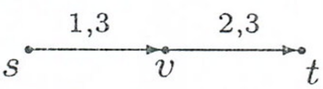
Tegyük fel, hogy a $c(u, v)$ kapacitások mellett – amelyek felülről korlátozzák az u-ból közvetlenül v-be áramló folyamot – *alsó korlátjaink* is vannak az $f(u, v)$ mennyiségekre. Az ilyen hálózatokat formálisan egy (G, s, t, c, k) alakú ötössel írhatjuk le. Ebben az első négy összetevő értelmezése a régi. A k a G élein értelmezett, nemnegatív értékű függvény, ami az élekhez rendelt alsó korlátokat mondja meg. A korábbi folyam-definíció kiegészítéseként megköveteljük, hogy $k(u, v) \leq f(u, v)$ is teljesüljön a G minden $u \to v$ élére.

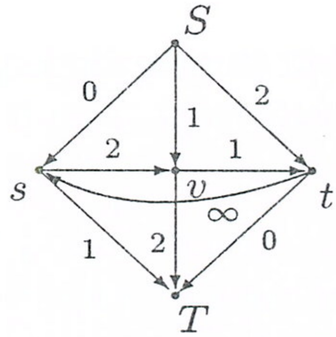
Az egyszerűbb modellnél a folyamok létezése felettébb jámbor kérdésnek bizonyult: a nulla-folyam minden hálózaton eleget tett a definíciónak. Az új modellhez nyilvánvalóan nincs ilyen univerzális választás.

Most megmutatjuk, hogy (régi értelemben vett) *maximális folyamok segítségével hatékonyan eldönthető, hogy egy $\mathcal{H} = (G, s, t, c, k)$ alakú hálózathoz létezik-e folyam.*

Evégből egy alsó korlátok nélküli $\mathcal{H}'$ hálózatot készítünk. Kényelmi okokból tegyük fel, hogy G-ben nincs $s \to t$ és $t \to s$ él. Vegyünk fel a $G = (V, E)$ pontjai mellé egy új forrást és egy új nyelőt; legyenek ezek S és T. A $\mathcal{H}'$ hálózat ponthalmaza $V \cup \{S, T\}$ lesz. A G éleit megtartjuk, de új kapacitásokat rendelünk hozzájuk. Az $(u, v) \in E$ él $c'(u, v)$ kapacitása a $\mathcal{H}'$ hálózatban legyen $c'(u, v) := c(u, v) - k(u, v)$. (Nyilván feltehetjük, hogy $c'(u, v) \geq 0$; ellenkező esetben nem létezhet folyam az eredeti hálózatban, mivel már az $u \to v$ élre kirótt feltételek sem teljesíthetők.) Adjunk még a $\mathcal{H}'$-höz egy $S \to v$ és egy $v \to T$ élet minden $v \in V$-re. Az $S \to v$ él kapacitása legyen $c'(S, v) := \sum_{(u,v)\in E} k(u, v)$, vagyis a v-be érkező G-beli élek alsó korlátjainak az összege. Hasonlóan, legyen $c'(v, T) := \sum_{(v,w)\in E} k(v, w)$, a v-ből kimenő G-beli élek korlátjainak az összege. Végül csapjunk még a hálózathoz egy ∞ kapacitású $t \to s$ élet.

A következő rajzon az átalakítást szemléltetjük. A baloldali gráf éleire írtuk az alsó korlátokat és a kapacitásokat. Így például $k(s, v) = 1$ és $c(v, t) = 3$. Mellette a megfelelő $\mathcal{H}'$ hálózat szerepel; az éleken a c' kapacitásokat tüntettük fel.





Tétel: *A* $\mathcal{H} = (G, s, t, c, k)$ *hálózatban akkor és csak akkor létezik folyam, ha a* $\mathcal{H}'$ *hálózat (S-ből T-be menő) maximális folyamának az értéke* $\sum_{(u,v)\in E} k(u, v)$.

Megjegyezzük, hogy a $\mathcal{H}'$-re vonatkozó állítás egyenértékű azzal, hogy egy maximális folyamra nézve az $S \to v$ és a $v \to T$ élek mind telítettek. Az $\{S\}$, $V \cup \{T\}$ és a $V \cup \{S\}, \{T\}$ vágások kapacitása ugyanis éppen $\sum_{(u,v)\in E} k(u, v)$.

Bizonyítás: Tegyük fel először, hogy van egy f folyamunk $\mathcal{H}'$-ben, aminek az értéke $\sum_{(u,v)\in E} k(u, v)$. Ebből egy g folyamot konstruálunk $\mathcal{H}$-ban. A G gráf minden (u, v) élére végezzük el a következőket: vonjunk le $k(u, v)$-t $f(S, v)$-ből és $f(u, T)$-ből és adjunk ugyanennyit $f(u, v)$-hez.

A módosítások során megtartottuk a Kirchoff-törvényt a G csúcsaiban (az átmenő folyamból ugyanannyit vontunk le, mint amennyit hozzáadtunk). A módosítások után az S-hez és T-hez illeszkedő éleken áramló mennyiség 0, így ezeket figyelmen kívül hagyhatjuk. Ha még a (t, s) élről és a rajta átmenő folyamról is elfeledkezünk, akkor azt kapjuk, hogy a G élein értelmezett $g(u, v) := f(u, v) + k(u, v)$ függvény s és t kivételével mindenütt eleget tesz a Kirchofftörvénynek. Legyen most $u \to v$ egy éle G-nek. A

$$0 \leq f(u, v) \leq c'(u, v) = c(u, v) - k(u, v)$$

egyenlőtlenségekből látjuk, hogy $k(u, v) \leq f(u, v) + k(u, v) \leq c(u, v)$. Itt a középen álló mennyiség nem más, mint $g(u, v)$. Tehát g tényleg egy folyam a $\mathcal{H} = (G, s, t, c, k)$ hálózatban.

A megfordítást úgy bizonyítjuk, hogy egy $\mathcal{H}$-beli g folyamból egy alkalmas f-et konstruálunk. Ezt lényegében az előző eljárás fordítottjával tehetjük meg. Először egy f' függvényt definiálunk a $\mathcal{H}'$ élein. Legyen $f'(t, s) = |g|$, $f'(u, v) = g(u, v)$, ha $u \to v \in E$ és $f'(S, v) = f'(v, T) = 0$, ha $v \in V$. Az így kapott f' a $V \cup \{S, T\}$ minden pontjában betartja a Kirchoff-törvényt. Ezután módosítjuk f'-t. Egymás után véve az $(u, v) \in E$ éleket adjunk $k(u, v)$-t $f'(S, v)$-hez és $f'(u, T)$-hez, és vonjunk le $k(u, v)$-t az $f'(u, v)$-ből. Az előzőek mintájára könnyű megmutatni, hogy az eredményül kapott f egy maximális folyam a $\mathcal{H}'$ hálózatban, amelyre $|f| = \sum_{(u,v)\in E} k(u, v)$. Ennek a részleteit az olvasóra hagyjuk.

□

A tételből következik, hogy folyamalgoritmus alkalmazásával $O(en^2)$ lépésben eldönthető, hogy van-e folyam a (G, s, t, c, k) hálózatban (n a G pontjainak, e pedig az éleinek a száma).

Feladat: Tegyük fel, hogy a $\mathcal{H}'$ konstrukciójában a $t \to s$ élhez ∞ helyett egy $d > 0$ kapacitást rendelünk. Mutassuk meg, hogy az így módosított $\mathcal{H}'$-höz pontosan akkor létezik $\sum_{(u,v)\in E} k(u,v)$ értékű maximális folyam, ha $\mathcal{H}$-ban van olyan g folyam, amelyre $|g| \leq d$.

Feladat: Tegyük fel, hogy a $\mathcal{H} = (G, s, t, c, k)$ hálózathoz tartozó kapacitások és alsó korlátok mind egészek. Legyen $d \in \mathbb{Z}$. Igazoljuk, hogy ha a $\mathcal{H}$-hoz van olyan g folyam, amelyre $|g| \leq d$, akkor olyan g^* folyam is van, amelyre $|g^*| \leq d$ szintén teljesül, és ezenfelül a $g^*(u,v)$ számok mind egészek. (A $\mathcal{H}, d$ párhoz tartozó $\mathcal{H}'$ hálózat kapacitásai mind egészek. Ennek egy egész értékkészletű maximális folyamából a tételbeli eljárást követve kaphatunk egész értékkészletű g^*-ot.)

Egy ütemezési feladat

Zárópéldaként egy ütemezési feladatot körvonalazunk, amihez – talán kissé váratlanul – egy alsó korlátokkal rendelkező hálózat ad megoldást. Tegyük fel, hogy egy légitársaság a $J_1, J_2, \ldots, J_m$ járatokat szeretné üzemeltetni, és d darab azonos típusú (mondhatni: csereszabatos) repülőgépe van erre a célra. Minden J_i, J_j járatpárra ismert, hogy van-e elég idő arra, hogy a J_i teljesítése után egy gép felkészüljön a J_j repülésére. Ha J_i, J_j-re a válasz igenlő, akkor azt mondjuk, hogy *J_j követheti J_i-t.*

A követhetőségi viszonyok – melyek egy irányított gráfot jelentenek a járatok halmazán – ismeretében szeretnénk eldönteni, hogy megvalósíthatók-e a járatok legfeljebb d repülőgéppel.

Egy olyan hálózatot készítünk, amelyben a J_i légijáratnak két csúcs, i és i', valamint az $i \to i'$ él felel meg ($1 \leq i \leq m$). Ha J_j követheti J_i-t, akkor vezessünk irányított élet i'-ből j-be. Vegyünk még fel egy s forrást és egy t nyelőt, és adjuk a hálózathoz az $s \to i$ és $i' \to t$ éleket ($1 \leq i \leq m$). Az összes él kapacitása legyen 1. Az $i \to i'$ alakú élek alsó korlátja legyen 1, a többi élé pedig 0.

Állítás: *A $J_1, J_2, \ldots, J_m$ járatok akkor és csak akkor teljesíthetők legfeljebb d géppel, ha a hálózathoz van olyan g folyam, amelyre $|g| \leq d$.*

Bizonyítás: Tegyük fel először, hogy létezik egy g folyam a hálózaton, amelyre $|g| \leq d$. Az előző feladat alapján feltehető, hogy g minden élen egész számot vesz fel. Ebből a kapacitások és a korlátok figyelembe vételével adódik, hogy $g(u,v) \in \{0,1\}$ minden $u \to v$ irányított élre. Vegyük szemügyre azt a részgráfot, aminek

az élei az egységnyi folyamot szállító élek. Ez – az élidegen utaknál látott módon – felbontható legfeljebb d darab

$$(\dagger) \qquad s \to i_1 \to i_1' \to i_2 \to i_2' \to \cdots \to i_l \to i_l' \to t$$

út egyesítésére. Két ilyen útnak s és t kivételével nincs közös pontja. A (†) út létezése azt jelenti, hogy a $J_{i_1}, J_{i_2}, \ldots, J_{i_l}$ járatok egy géppel teljesíthetők. A $k(i, i') = 1$ feltételek miatt minden (i, i') él szerepel valamelyik ilyen úton. Tehát legfeljebb d géppel tényleg minden járat lebonyolítható.

Fordítva, tegyük fel, hogy van egy legfeljebb d gépet használó, minden járatot teljesítő ütemezésünk. Nézzük ebben egy gép egymás utáni feladatait. Ha ezek a $J_{i_1}, J_{i_2}, \ldots, J_{i_l}$ járatok (ebben a sorrendben), akkor a (†) út éleit a hálózat tartalmazza. Ezen az úton egységnyi folyamot küldhetünk s-ből t-be. Ezt minden gépre elvégezve egy legfeljebb d értékű g folyamot kapunk. A kapacitásfeltételek teljesülnek, mert az így kapott (†) utaknak s és t kivételével nincs közös élük, hiszen egy járat egy géphez tartozik. Az (i, i') élekre kiszabott $g(i, i') \geq 1$ feltételeket is megtartjuk, mert az ütemezésben minden járatnak van gazdája. Megállapíthatjuk immár, hogy g eleget tesz a követelményeknek. □

Feladat: Mutassuk meg, hogy a követhetőségek gráfjának és d-nek az ismeretében $O(m^4)$ lépésben eldönthető, hogy van-e a $J_1, J_2, \ldots, J_m$ járatoknak legfeljebb d géppel való ütemezése. (Az előzőek segítségével alakítsuk át a feladatot maximális folyam meghatározásává; ennek megoldására használjuk Dinic módszerét.)

7.

Turing-gépek

Ha az emberi agy olyan egyszerű lenne, hogy megérthessük, akkor túl egyszerűek lennénk ahhoz, hogy ezt megtehessük. AMERIKAI FALFIRKA

Ebben a fejezetben elsődleges célunk az *algoritmus* fogalmának egy lehetséges pontosabb körülhatárolása. A fő eszközünk ebben egy egyszerű számítási modell, az Alan M. Turing[1] angol kutató által kidolgozott Turing-gép lesz. A modell a számítógép, a program egyszerűsített és bizonyos tekintetben idealizált változatának tekinthető. Egyszerűsége ellenére igen erőteljes konstrukció: minden olyan probléma, ami megoldható valamilyen algoritmus segítségével, kezelhetőnek bizonyult Turing-géppel is. A fejezet elején a Turing-gépekkel kapcsolatos egyszerű fogalmak, eredmények szerepelnek, amiket aztán később gyakran fogunk használni.

Sokáig úgy gondolták, hogy minden algoritmikus problémára van megoldás. Ha elég szívósan próbálkozunk, akkor előbb-utóbb legyűrjük a problémát. A harmincas években, főként A. Church, K. Gödel és A. Turing munkássága nyomán kiderült, hogy ez távolról sem igaz: vannak olyan feladatok, amelyekre nem létezik

[1] Alan Mathison Turing (1912-1954) egyike volt századunk legjelentősebb gondolkodóinak. Sikeresen foglalkozott matematikával, a számítások elméletével és gyakorlatával, de filozófiával és agykutatással is. Munkásságának egyik legnevezetesebb része a Turing-gép, és ehhez kapcsolódóan a kiszámíthatóság elméletének alapjai. Ezekről a fejezetben részletesen is lesz szó. A II. világháború alatt ő vezette azt a csoportot, amelyik feltörte a német tengeralattjárók által használt *Enigma*-kódot. Ennek óriási szerepe volt abban, hogy a szövetséges konvojok a normandiai invázióra készülve zavartalanul közlekedhettek az atlanti vizeken. A háború után egyik vezéralakja volt az első brit számítógép, az *ACE* tervezőgárdájának. Őt tekinthetjük a mesterséges intelligencia mint kutatási irány megalapozójának is. Ilyen tárgyú gondolatai – köztük a nevezetes *Turing-teszt* – ma is nagy hatásúak. Megemlítjük még, hogy a sokak által számítástudományi-számítástechnikai Nobel-díjnak tekintett kitüntetés neve: *ACM Turing Award*.

megoldó módszer. Ennek a kérdéskörnek, a *kiszámíthatóság elméletének* az elemei jelentik a következő témát, amelyet tárgyalni fogunk. Megismerkedünk néhány nevezetes algoritmikusan eldönthetetlen feladattal (Megállási probléma, Hilbert 10. problémája stb.).

Korábban már foglalkoztunk információtömörítő módszerekkel. A *Kolmogorov-bonyolultság* segítségével itt azt tanulmányozzuk, hogy a véges jelsorozatok mennyire nyomhatók össze valamilyen algoritmussal. Ki fog derülni, hogy vannak nem vagy csak alig összenyomható szavak, sőt, bizonyos értelemben az ilyen szavak a tipikusak. A módszer erejét alkalmazásokkal illusztráljuk.

A fejezet végén egy másik gépmodellel, a *közvetlen elérésű géppel* ismerkedünk meg. A Turing-gépnek mint elméleti eszköznek az erénye az egyszerűség. Ugyanezen tulajdonsága miatt viszont túl nehézkes ahhoz, hogy a segítségével igazi algoritmusokat írjunk le. Programozási eszközként nagyobb büntetést jelentene a a legprimitívebb gépi kódnál is. A közvetlen elérésű gép számítóerő dolgában, tehát a megoldható feladatok összességét tekintve egyenértékű a Turing-gépekkel. Másfelől a közvetlen elérésű gép sokkal jobban hasonlít az igazi számítógépek architektúrájára. Ezáltal módot ad arra, hogy szabatos, ugyanakkor gyakorlatias módon definiáljuk a programok időköltségét.

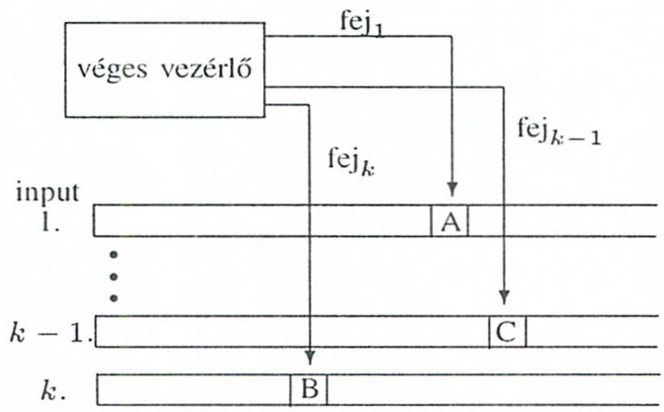
7.1. A Turing-gép fogalma

Hosszabb távú törekvésünk, hogy az *algoritmus* fogalmának egyik lehetséges pontos megközelítését adjuk. Ezt nagyjából úgy fogjuk tenni, hogy definiálunk egy számítási modellt, egy gépet, és azt tekintjük algoritmusnak, amit ezzel a géppel meg lehet valósítani. Ez a gép a *Turing-gép* lesz. A Turing-gép kevéssé hasonlít a ma használatos számítógépekre. Egy igen egyszerű központi részből és néhány *szalagegységből* áll. A központi rész, amit *véges vezérlőnek* is neveznek, a gép munkája során véges sok lehetséges *állapot* valamelyikében van. A szalagok cellákra vannak osztva. (Ezeket szokás még mezőknek, olykor négyzeteknek is nevezni.) Egy cellán a gépre jellemző rögzített véges jelkészlet, a *szalagjelek* halmazának egy eleme lehet. A szalagokat egyirányban végtelennek tekintjük. Ez az egyetlen sajátosság, amiben a Turing-gép erősebb a valóságos gépeknél. Mindegyik szalaghoz tartozik egy *író–olvasó fej*, amely írhatja/olvashatja a fej alatti mezőt. A fejek mindkét irányban léphetnek a szalagjuk mentén. A szalagok adják a Turing-gép adattárolási lehetőségeit; nincs külön belső memória és háttértár, mint az igazi számítógépeknél.

A gép bemenete egy szalagjelekből álló véges sorozat, ami a számítás megkezdésekor az egyik szalagra van írva. A gép számítása *lépések* egymásutánjából áll. A gép egy lépésében elolvassa az éppen a fejek alatt levő jeleket, majd ezektől

és a központi egység belső állapotától függően „cselekszik". Az egyik lehetőség, hogy megáll, azaz nem tesz semmit. A másik lehetőség, hogy egy-egy jelet ír a fejek alatti cellákba, majd egy mezővel jobbra vagy balra lépteti, esetleg helyben hagyja a fejeket, és végül a központi egység új állapotot vesz föl.

Ha a gép megállt, akkor megnézhetjük, hogy mi a számítás eredménye. A központi egység végső állapotát vagy pedig az egyik szalag tartalmát fogjuk eredményként értelmezni. Előfordulhat, hogy az adott bemenetre a gép sohasem áll meg. Ezt a jelenséget az igazi gépek világából vett, de itt a végtelen szalagok miatt kissé pongyola szóhasználattal *végtelen ciklusnak* is nevezik. A következő ábra a Turing-gép felépítését szemlélteti.



Ezek után ismertetjük a Turing-gép formális definícióját. Legyen k egy pozitív egész. Egy *k-szalagos Turing-gép* egy hetessel jellemezhető:

$$M = (Q, T, \text{ü}, I, q_0, F, \delta),$$

ahol

Q : egy véges halmaz, az M gép *belső állapotainak* halmaza. A gép a működése során mindig valamelyik $q \in Q$ állapotban van.

T : egy véges halmaz, a *szalagjelek* halmaza. Egy szalagmezőn mindig egy T-beli jel van.

ü : egy kitüntetett szalagjel, az *üresjel*. A bemenetként felírt jeleken és a gép számítása során kiírt jeleken kívül minden szalagcellában *ü* van. Az üresjelet tekinthetjük úgy, mint a még nem használt mezők tartalmát.

I : $I \subseteq T \setminus \{\text{ü}\}$ az *input jelek* vagy *bemenő jelek* halmaza, más szóval az input *abc*. A gépnek adott bemenetet az I elemeiből kell összeállítani. Az üresjel nem lehet input jel.

q_0 : $q_0 \in Q$ a *kezdő állapot*. A gép a munka megkezdése előtt a q_0 állapotban van.

F : $F \subseteq Q$ az *elfogadó állapotok* halmaza. Bizonyos számításoknál csak egy kétértékű döntést várunk a géptől (pl. az igen-nem választ igénylő feladatok esetében). Ekkor csak azt nézzük, hogy a gép milyen állapotban állt meg. Az egyik választ az jelenti, hogy ez a végállapot F-beli, a másikat pedig az, hogy nem F-beli. Ezzel összhangban a $Q \setminus F$-be tartozó állapotokat *elutasító állapotoknak* is nevezik.

δ : A $\delta : Q \times T^k \longrightarrow Q \times (T \times \{jobb, bal, helyben\})^k$ egy parciálisan értelmezett függvény, a gép *átmenetfüggvénye*. Az átmenetfüggvény tekinthető a gép programjának. A δ függvény mondja meg, hogy ha a gép a q állapotban van, és az i-edik fej alatt levő szalagjel a_i ($1 \leq i \leq k$), akkor mit kell lépni. Ennek megfelelően a $\delta(q; a_1, a_2, \ldots, a_k)$ nincs értelmezve, ha ez a lépés a megállás. A többi esetben a $\delta(q; a_1, a_2, \ldots, a_k)$ érték $2k + 1$ összetevőből áll. Ezek: a gépnek a lépés utáni $q' \in Q$ állapota; továbbá az i-edik szalagra kiírt b_i szalagjel, valamint az i-edik fej elmozdulása ($1 \leq i \leq k$). Az elmozdulás legfeljebb mezőnyi, ezért leírható a *jobb*, *bal* és *helyben* lehetőségek egyikeként. A δ függvényről még feltesszük, hogy az F-beli állapotokon nincs értelmezve.

A δ parciális függvény lényegében egy (nem mindenütt kitöltött) táblázatként fogható fel. Ez a táblázat írja le a gép programját. A Turing-gép a korábbiaknál kicsit pontosabban olyan gépnek tekinthető, amely egyetlen program futtatására képes.

Az I input *abc* elemeiből álló véges sorozatokat *szavaknak* nevezzük. Az I jeleiből (betűiből) alkotott szavak halmazát I^*-gal jelöljük. I^* részhalmazait *nyelveknek* nevezzük. Egy $L \subseteq I^*$ nyelv elemeit az L nyelv szavainak nevezzük.

A Turing-gép első szalagját *input szalagnak* is nevezzük, mivel azon lehet a bemenetet megadni. Bizonyos esetekben hasznos, ha kijelölünk egy *output szalagot* is. Ez annyit tesz, hogy a gép megállása után az eredményt ennek a szalagnak az elejéről olvassuk le. Az output szalag megadása nem több, mint a végeredmény megjelenési helyére vonatkozó előírás.

A Turing-gép működése

A gép a működés előtt *kezdőhelyzetben* van. Ez azt jelenti, hogy a gép állapota a q_0 kezdőállapot, az író-olvasó fejek a szalagjaik első cellájára mutatnak, az $s \in I^*$ bemenet az első szalag elejére van írva, az összes többi mezőn az *ü* üresjel található.

A gép a kezdőhelyzetből indulva lépéseket tesz. A lépések tartalmát jelentő változásokat a δ függvény írja le: a lépés előtti állapotnak és a fejek alatti jeleknek megfelelő δ-érték adja meg az új állapotot, a kiírandó jeleket és a fejmozgásokat. Ha a gép egy olyan helyzethez ér, amelyre δ nincs értelmezve, akkor megáll. Ez történik egyebek között akkor, ha a gép elfogadó (azaz F-beli) állapotban van.

A Turing-gép számításának eredménye
A Turing-gépeket kétféle üzemmódban fogjuk tekinteni. Egyfelől *nyelvek felismerésére,* másfelől *függvények kiszámításra* használjuk őket. Az utóbbi típusú feladatok eléggé természetesek a számítástechnikában, a velük való foglalkozás így nem igényel különösebb indoklást. Ami a nyelvek felismerését illeti, ezek olyan feladatoknak tekinthetők, ahol a számítás eredménye egyetlen bit: egyetlen eldöntendő kérdésre kell választ adni. Az ilyen feladatok elméleti szempontból egyszerűbben megfoghatók, ugyanakkor elég gazdag a struktúrájuk ahhoz, hogy segítségükkel a számítások általános tulajdonságairól képet kaphassunk.

Definíció (Turing-gép által felismert nyelv):
Az M Turing-gép által felismert L_M nyelv *azokból az $s \in I^*$ szavakból áll, amelyekkel mint bemenetekkel elindítva az M megáll, mégpedig elfogadó (azaz F-beli) állapotban.*

Ez a fogalom tulajdonképpen az *igen–nem kérdések* eldöntésének felel meg: képzeljük el, hogy van egy általános kérdésünk, ami az I^* minden egyes szavára értelmes, és a válasz egy szó esetén igen vagy nem. Például a kérdés lehet az, hogy az $s \in I^*$ szó tartalmazza-e a 0 jelet. Egy ilyen kérdésnél azok a szavak, amelyekre a válasz igenlő – az ún. igenpéldányok –, egy $L \subseteq I^*$ nyelvet alkotnak. Ha $L = L_M$, vagyis az igenpéldányokból álló nyelv éppen az M által felismert nyelv, akkor úgy tekinthetjük, hogy az M gép gyenge értelemben eldönti a kérdést. Ha $s \in I^*$ igenpéldány, akkor ezt M véges sok lépésben megállapítja azzal, hogy megáll elfogadva s-et. Ha s nem egy igenpéldány, akkor cizelláltabb a kép. Lehet, hogy M ekkor is megáll, szükségképpen elutasító állapotban, amit úgy értelmezhetünk, hogy ebben az esetben is sikerült megválaszolnia a kérdést. A definíció megengedi viszont azt is – és ez a *gyenge* magyarázata –, hogy a gép $s \notin L_M$ esetén sose álljon meg. Az M tehát általában véve egy fél algoritmusnak tekinthető az L-nek megfelelő kérdés megválaszolására. Az L_M szavaira jól működik, az $I^* \setminus L_M$ szavainál viszont végtelen ciklusba keveredhet.

Definíció (Turing-gép által kiszámított függvény):
Legyen M egy kitüntetett output szalaggal rendelkező Turing-gép. Az M által kiszámított $f_M : I^* \to I^*$ parciális függvényt *így értelmezzük:* $f_M(s) = w$*, ha M*

az $s \in I^*$ inputtal indulva megáll, és megállás után az output szalagon a $w \in I^*$ szó szerepel az üresjelek óceánja előtt.

Az f_M függvény tehát csak azokra az $s \in I^*$ szavakra értelmezett, amelyekkel mint bemenetekkel az M gép véges sok lépés megtétele után megáll. Ennél a számítási módnál közömbös, hogy a megállás elfogadó vagy elutasító állapotban történt-e. Az $f_M(s)$ eredmény pedig a kitüntetett szalag legnagyobb kezdődarabja, ami csupa I-beli betűből áll.

Példák: 1. A következő igen egyszerű egyszalagos M gép bizonyos értelemben a hárommal való oszthatóságot dönti el. Legyen tehát $k = 1$, és

$$Q = \{q_0, q_1, q_2, q_v\},\ T = \{0, 1, ü\},\ I = \{0, 1\},\ F = \{q_v\}.$$

A gépnek négy állapota van, ebből egy elfogadó állapot. A bemenetet a 0 és 1 jelekből lehet összeállítani. Mivel M-nek egy szalagja van, a δ függvény *(állapot, szalagjel)* alakú párokhoz rendel *(állapot, szalagjel, fejmozgás)* összetételű hármasokat. Legyen ezek után δ a következő:

$$\begin{aligned} \delta(q_0, 1) &= (q_1, 1, jobb),\ \ \delta(q_0, ü) = (q_v, ü, helyben), \\ \delta(q_1, 1) &= (q_2, 1, jobb),\ \ \delta(q_2, 1) = (q_0, 1, jobb). \end{aligned}$$

Tegyük még fel, hogy más párokra δ nincs értelmezve. Az M átmeneteit nézve megállapíthatjuk, hogy ha 0 jelet olvas, akkor megáll elutasító, azaz nem elfogadó állapotban. Nincs ugyanis a δ értelmezési tartományában olyan pár, ami a 0 szalagjelet tartalmazza. Ugyanez a helyzet az *ü* üresjellel is, kivéve, ha az állapot q_0, mely esetben M a fejet helyben hagyva átmegy a q_v elfogadó állapotba. Érdekesebb, ami egyes olvasásakor történik. Ha a gép állapota ekkor q_i, akkor a gép érdemi írás nélkül (ugyanazt írja vissza amit olvasott) jobbra lép, és átmegy a q_{i+1} állapotba; itt az összeadás modulo 3 értendő. Figyelembe véve még azt is, hogy M a q_0 állapotból indul, mindezekből arra jutunk, hogy M pontosan azokat a szavakat fogadja el, amelyek csupa egyesekből állnak, és a hosszuk osztható hárommal. Az M által felismert L_M nyelv tehát

$$L_M = \{1^n : n \text{ hárommal osztható természetes szám}\}.$$

Itt 1^n az n darab egyesből álló szó jelölése; speciálisan $1^0 = ü$.

2. Az itt szereplő egyszalagos M' gépet mindkét üzemmódjában megnézzük. Meghatározzuk az $L_{M'}$ nyelvet és az $f_{M'}$ függvényt is. Utóbbihoz output szalag is kell. Nincs sok választásunk: az egy szem szalagot használjuk erre a célra is. Legyen

$$Q = \{q_0, q_v\},\ T = \{0, 1, ü\},\ I = \{0, 1\},\ F = \{q_v\}$$

és

$$\delta(q_0, 0) = (q_0, 0, helyben),\ \delta(q_0, 1) = (q_0, 1, jobb),\ \delta(q_0, ü) = (q_v, 1, helyben).$$

Másutt δ nem definiált. Az M' a q_0 belső állapotban maradva lépdel jobbra a szalag mentén, amíg 0 vagy *ü* jelet nem talál. Az előbbi esetben végtelen ciklusba kerül, az utóbbiban pedig még egy 1-est ír az input után, majd megáll elfogadó állapotban. A gép tehát a csupa egyesekből álló szavakat fogadja el: $L_{M'} = \{1^n; n \geq 0\,\text{egész}\}$. Az M' csak az $L_{M'}$ szavaira áll meg, így ez a nyelv az $f_{M'}$ parciális függvény értelmezési tartománya. A gép az 1^n bemeneten az 1^{n+1} eredményt adja, vagyis $f_{M'}(1^n) = 1^{n+1}$. Az M' gépet tekinthetjük az $f(n) := n + 1$ függvényt kiszámító algoritmusnak.

Feladat: Az 1. példabeli M gép az unárisan ábrázolt n számról eldönti, hogy az hárommal osztható-e. Szerkesszünk olyan N Turing-gépet, amelyik a *binárisan* megadott n bemenettel teszi ugyanezt.

Talán az előző példák hihetővé teszik, hogy tényleg lehet számításokat végezni Turing-gépekkel. A tényleges gépek és a Turing-gépek számítóereje közötti kapcsolat további hangsúlyozásául megjegyezzük, hogy utóbbiak leírhatók programmal. Egy lehetséges megoldás, hogy helyet foglalunk a szalagok véges darabjainak. Ezeken nyilvántartjuk a fejek helyzetét; legyenek ezek az $f_1, f_2, \ldots, f_k$ változókban. A gép mindenkori belső állapotát pedig a q változóban tároljuk. Ezután egy végtelen ciklust szervezünk, aminek magjában a következők szerepelnek:

- Az olvasófejek alatti szalagjelek beolvasása az $x_1, \ldots, x_k$ változókba.

- a δ táblázat minden bejegyzésére egy programrészlet a következő minta szerint: a $\delta(q_i; a_1, a_2 \ldots, a_k) = (q_j; (b_1, bal), (b_2, helyben), \ldots, (b_k, jobb))$ érték megfelelője legyen

 if $q = q_i$ **and** $x_1 = a_1$ **and** $x_2 = a_2$ **and** $\ldots$ **and** $x_k = a_k$ **then begin**
 $q := q_j$;
 $x_1 := b_1;\ x_2 := b_2;\ \ldots;\ x_k := b_k$;
 Az x_i tartalmának kiírása az f_i változókkal azonosított helyekre
 $(i = 1, 2, \ldots, k)$;
 $f_1 := f_1 - 1;\ \ldots\ ; f_k := f_k + 1$;
 end;

- Megállás abban az esetben, ha δ az adott helyen nincs értelmezve.

Az irodalomban a Turing-gépeknek többféle, a részletekben kisebb-nagyobb eltéréseket mutató meghatározásával találkozhatunk. Ilyen eltérés lehet, hogy a szalagokat mindkét irányban végtelennek vesszük. Az eltérések azonban elméleti szempontból nem jelentősek; a különböző géptípusok egyenértékűsége némi munkával igazolható. Könnyen átalakítható például egy Turing-gép ugyanazt a nyelvet felismerő, de csak egyetlen elfogadó állapottal rendelkező géppé. Ennek meggondolását az olvasóra bízzuk.

7.2. Idő- és tárigény

A Turing-gép mint modell egyik erénye, hogy segítségével elég egyszerűen vizsgálhatjuk a számítások idő- és tárigényét. Az M Turing-gép *számolási ideje* az s inputon a megállásáig végrehajtott lépések száma, *tárigénye* pedig a felhasznált (olvasott) szalagcellák száma. Ha a gép input szalagja csak olvasható, akkor előírhatjuk, hogy ez a szalag ne számítson bele a tárigénybe. Hasonló a helyzet a csak írásra szolgáló output szalaggal (amin sosem léphetünk balfelé). Az a megfontolás áll emögött, hogy kizárólag az érdemi munka helyigényét mérjük, és ne foglalkozzunk a bemenet olvasásának és az eredmény írásának (elkerülhetetlenül fellépő) költségével. Úgy is fogalmazhatunk, hogy csupán a *munkaszalagokon*, az írásra és olvasásra is használatos szalagokon felhasznált hely számít.

A hárommal való oszthatóságot eldöntő M gép számolási ideje az 1^n bemeneten $n+1$. A tárigény függ attól, hogy miként szemléljük a szalagot. Tekinthetjük csak olvasható input szalagnak, hiszen a gép sohasem írja (ami azt jelenti, hogy mindig a régi értéket írja a fej alatti mezőbe). Ekkor a tárigény minden bemeneten 0. Ha viszont a szalagot munkaszalagnak vesszük, akkor az 1^n bemeneten a tárigény $n+1$ lesz. Az $f(n) := n+1$ függvényt kiszámító M' gép számolási ideje az 1^n szóra $n+1$, az 110 szóra pedig végtelen, hiszen a 0 olvasása után M' végtelen ciklusba esik. A szalagot tekinthetjük csak írható output szalagnak, mert a gép sohasem lép balra. Ekkor a tárigény minden bemeneten 0. Ha viszont munkaszalagként szemléljük, akkor 1^n-nél a tárigény $n+1$, az 110 bemeneten pedig 3.

Szeretnénk beszélni az algoritmusok (Turing-gépek) sebességéről, illetve tárfelhasználás szerinti hatékonyságáról. Ebben túlságosan elaprózott megközelítés lenne, ha minden $s \in I^*$ szóra külön vizsgálnánk a lépésszámot vagy a tárigényt. Ennél egyszerűbb, de még mindig elég informatív megoldást kapunk, ha a bemenet hosszának függvényében vizsgáljuk ezeket az igényeket.

Jelölje $T_M(n)$ az M gép maximális számolási idejét az n jelből álló bemeneteken. Az n hosszú szavakon a maximális tárigényt $S_M(n)$-nel jelöljük.

Ha van olyan n jelből álló $s \in I^*$ szó, amellyel elindítva M nem áll meg véges sok lépés után, akkor $T_M(n)$ értékét végtelennek tekintjük. Hasonló mondható az S_M függvényről is.

A hárommal való oszthatóságot tesztelő M gépre $T_M(n) = n + 1$, ha pedig a szalagját munkaszalagnak vesszük, akkor $S_M(n) = n + 1$. Ami az M' gépet illeti, $T_{M'}(n) = \infty$, ha $n > 0$, hiszen a nullát tartalmazó bemeneteken a gép nem áll meg. Ha M' szalagja munkaszalag, akkor $S_{M'}(n) = n + 1$.

A T_M és S_M függvények segítségével lehetőség nyílik arra, hogy algoritmusokat hatékonyság szempontjából összehasonlítsunk. Ha például M és N két Turing-gép, melyekre $T_M(n) < T_N(n)$ teljesül minden elég nagy n-re, akkor az M algoritmust gyorsabbnak mondhatjuk az N algoritmusnál. Ily módon beszélhetünk arról, hogy egy adott feladatot megoldó két módszer közül az egyik hatékonyabb, mint a másik.

Algoritmikus problémák megoldása során igyekszünk minél jobb módszereket találni. Ha például az L nyelv időben hatékony felismerése a feladat, akkor olyan M algoritmust keresünk, amelyre a $T_M(n)$ függvény elég lassan nő. Természetes törekvésnek tűnhet, hogy ilyen értelemben a legjobb módszert keressük. Legjobb módszer azonban nem feltétlenül létezik, amint azt a következő tétel mutatja.

Tétel (gyorsítási tétel): *Van olyan L nyelv, amelyre igazak az alábbiak:*
1. Az L felismerhető egy olyan M Turing-géppel, melyre $T_M(n)$ véges minden n-re.
2. Tetszőleges, az L-et felismerő N Turing-géphez van olyan N' Turing-gép, amelyre $L = L_{N'}$ szintén teljesül, továbbá $T_{N'}(n) = O(\log T_N(n))$.

□

A bizonyítást mellőzzük. A tételbeli L nyelvet felismerő bármely algoritmus exponenciálisan felgyorsítható. Ezért nem létezhet a feladatra leggyorsabb algoritmus. A feladatok idő- és tárigényének értelmezésében tehát óvatosabban kell eljárni. A következő fejezet elején visszatérünk ehhez a gondolatkörhöz.

7.3. Néhány szimuláció

Ebben a szakaszban néhány olyan tényt ismertetünk, amelyek azt mondják, hogy bizonyos típusú Turing-gépek helyettesíthetők, *szimulálhatók* másfélékkel. A szimuláló gép egyenértékű az eredetivel abban az értelemben, hogy ugyanazt a nyelvet ismeri fel, illetve ugyanazt a függvényt számítja ki. Bemelegítőnek egy feladatot ajánlunk:

Feladat: Legyen M egyszalagos Turing-gép. Módosítsuk M-et úgy, hogy a kapott M' gép egyenértékű legyen M-mel, azaz $L_M = L_{M'}$, $f_M = f_{M'}$ teljesüljön;

továbbá az M' gépnek pusztán az aktuális állapotából kiderüljön, hogy mi volt a szalagjel, amit utoljára olvasott. (Az M' gép állapotai legyenek q_0 és a (q, a) alakú párok, ahol q_0, q az M állapotai, a pedig az M szalagjele. Az M gép δ átmenetfüggvényét terjesszük ki a párokra úgy, hogy az első komponensnek megfelelően utánozzuk M lépését, a második komponens pedig az éppen olvasott szalagjelnek feleljen meg.)

A feladattal azt szerettük volna érzékeltetni, hogy a belső állapotok segítségével fenntarthatunk a gépben egy rögzített méretű memóriát. Hasonló megoldással tárolhattuk volna mondjuk a fej helyzetének modulo 5 osztási maradékát vagy bármi más korlátos mennyiségű információt.

A szimulációs eredmények egy része olyasmit mond, hogy az egyszerűbb gépek viszonylag kézben tartható hatékonyságvesztéssel meg tudják tenni ugyanazt, amit a bonyolultabbak. A következő eredmény szerint az, amit el lehet végezni sok szalaggal, véghezvihető egyetlen szalaggal is. A tétel gyakran ad módot arra, hogy csak egyszalagos gépekre korlátozzuk a figyelmünket.

Tétel: *Legyen M egy k-szalagos Turing-gép. Van olyan egyszalagos M' Turing-gép, melyre*

$$L_M = L_{M'} \quad (\textit{vagy } f_M = f_{M'}), \textit{ továbbá}$$
$$T_{M'}(n) \leq 2T_M^2(n),$$
$$S_{M'}(n) \leq S_M(n) + n.$$

Bizonyítás: M' építésekor az M-hez képest alaposan felfújjuk a szalag abc-t, és megnöveljük a belső állapotok számát is. Az M' egyetlen szalagján $2k$ csík lesz. A $2i-1$-edik csík felel meg az M i-edik szalagjának, míg a $2i$-edik csíkban az i-edik fej helyén egy megkülönböztető jel, x szerepel. Az M' szalagjának egy mezején k eredeti szalagjelet és k darab fej-helyzetről tájékoztató jelet kódolunk.

1. szalag	D	$\cdots$	A	$\cdots$	B	$\cdots$
1. fej	ü	$\cdots$	x	$\cdots$	ü	$\cdots$
.						
.						
.						
k. szalag	D	$\cdots$	D	$\cdots$	B	$\cdots$
k. fej	ü	$\cdots$	ü	$\cdots$	x	$\cdots$

A rajzon egy oszlop felel meg az M' egyetlen szalagmezejének. Az ábrázolt helyzet az M szalagjainak azt az állapotát tükrözi, amikor az első és a k-adik szalag első mezején is D van, az első fej alatt A, a k-adik fej alatt pedig B olvasható, stb. Az M'-nek összesen $(2t)^k$ szalagjele van, ahol t az M szalagjeleinek a száma.

M' az M gép egy lépését egy legfeljebb $2T_M(n)$ lépésből álló menetben utánozza. Először a szalag elejéről indulva jobbra elmegy a legmesszebb levő x jelig, és közben leolvassa az x-ek felett található eredeti szalagjeleket. Ezeket a belső állapotaiban tárolja. (Ezt a feladatban javasolt módszerrel teheti.) A menetelés során egy hellyel jobbra mozdítja az útjába eső x-eket, kivéve az utolsó oszlopban levő(ke)t. Ebben a helyzetben a gép ismeri M állapotát és a fejei alatti jeleket, így meghatározhatja a M következő állapotát és a kiírandó jeleket.

Ezután M' legfeljebb egyet lép jobb felé, majd egyenesen visszabaktat a szalag elejére. Eközben kiírja az M fejeinek régi helyére a k darab jelet, amiket M írt volna, és az M fejmozgásainak megfelelően áthelyezi az x-eket. Utóbbiaknál hasznos, hogy korábban jobbra léptettük ezeket a jeleket. Így mindezek megoldhatók úgy is, hogy csak balfelé megyünk.

Ha n jelből álló bemenettel kezdjük a munkát, akkor egy meneteben az M' feje legfeljebb $T_M(n)$ lépést tesz jobbra, következésképpen legfeljebb ugyanennyit mehet balra. Az M egy lépését így nem több, mint $2T_M(n)$ lépéssel szimuláltuk. Az M' pontosan akkor álljon meg (fogadja el a bemenetet), ha M ezt teszi. Így a számolási idő $T_{M'}(n) \leq 2T_M(n)T_M(n) = 2T_M^2(n)$.

Ha függvényt számítunk, akkor az utolsó lépés visszatérő fázisában az M output szalagjának megfelelő csíkot kivéve mindenhova üresjelet írunk. Tesszük mindezt azért, hogy az eredmény az M bemenő betűivel legyen írva.

A tárra vonatkozó állításból ezután csak a $+n$ tag szorul némi magyarázatra. Ez azért szerepel, mert ha M-nek kitüntetett input szalagja volt, akkor a bemenet hosszát, ami n, nem számítottuk bele $S_M(n)$-be.

□

A következő tétel szerint a szimuláció idővesztesége kisebb lesz, ha egy helyett két szalagot engedünk meg. A bizonyítás az előbbinél bonyolultabb; nem részletezzük.

Tétel: *Az M k-szalagos Turing-géphez megadható olyan 2-szalagos M' Turing-gép, amely az előbbi értelemben szimulálja M-et,*

$$T_{M'}(n) \leq O(T_M(n) \log T_M(n)), \text{ és}$$

$$S_{M'}(n) \leq S_M(n) + n.$$

□

A most terítékre kerülő tény azt sugallja, hogy az algoritmusok időigényét sok esetben legfeljebb csak állandó szorzó erejéig érdemes nézni. A Turing-gép alkalmas megválasztásával ugyanis a multiplikatív konstans majdnem tetszőlegesen lefaragható.

Tétel: *Tegyük fel, hogy az M Turing-gép az L nyelvet ismeri fel, és $T_M(n) \leq cn$ teljesül egy $c > 0$ állandóval. Ekkor tetszőleges $\epsilon > 0$-ra van olyan L-et felismerő M' Turing-gép is, hogy alkalmas $n_0 \in \mathbb{N}$ számmal*

$$T_{M'}(n) \leq n(1+\epsilon), \text{ ha } n \geq n_0.$$

Bizonyítás: Az M' gépet úgy tervezzük, hogy az M sok egymás utáni lépését kevés lépésben utánozni tudja. Pontosabban egy kellően nagynak választott m számmal az fog teljesülni, hogy a régi gép m lépését az új legfeljebb 7 lépésben elvégzi. Osszuk fel evégből az M szalagjait m egymás utáni mezőből álló blokkokra. Egy ilyen blokk tartalmát az M' egyetlen szalagjelben tárolja. Eszerint, ha M-nek t szalagjele van, akkor az új gép t^m betűt használ. Az M'-nek eggyel több szalagja lesz, mint M-nek. Az elsőn, amit csak olvas, az M bemenete szerepel. Ennek tartalmát az érdemi munka előtt átkódolja egy másik szalagra, utána a régi bemenő szalaggal nem foglalkozik. A többi szalag – tömörebb kódolással – ugyanazt fogja tartalmazni, mint az M megfelelő szalagjai.

Nézzük most, hogy mi történik a szalagokkal az M gép m egymást követő lépése során! A lényeges észrevétel, hogy ami ezalatt végbemegy, az csak a fejeket tartalmazó blokkoktól és azok közvetlen szomszédaitól függ, hiszen a fejek legfeljebb m cellányira mozdulhatnak el. Változások is csak eme blokkokban lehetségesek.

M' tehát nagy vonalakban a következőket teszi: szalagonként három szomszédos jel (ami három blokkot jelent M-nél) megvizsgálása után M átmeneteinek ismeretében a „memóriájában" meglépi M következő m lépését. Ezután az eredménynek megfelelően felülírja a szomszédos mezőhármasokat, majd helyükre teszi a fejeket. A szükséges memória a feladatban vázolt eljárással képezhető.

Ezek szerint az m lépés szimulációjához a szalagok olvasása+írása elvégezhető egy bal–jobb–jobb–bal–bal lépéssorozatban (a szalagok elején ennél rövidebben is). Legfeljebb további két jobbralépés szükséges lehet, amikor az M helyzetének megfelelő blokkokra állítjuk az M' fejeit. Így M' legfeljebb 7 lépésben elvégzi az M gép m lépését. A bemenet átkódolása, majd a kódolt szalagon a fejnek a szalag elejére mozgatása összesen $n + \lceil \frac{n}{m} \rceil$ lépésben megtehető. Az M' lépéseinek számát így becsülhetjük:

$$T_{M'}(n) \leq n + \left\lceil \frac{n}{m} \right\rceil + 7 \left\lceil \frac{T_M(n)}{m} \right\rceil \leq n + \frac{n}{m} + \frac{7T(n)}{m} + 8 \leq$$

$$\leq n + \frac{n}{m} + \frac{7cn}{m} + \frac{8n}{n} \leq n(1 + \frac{1}{m} + \frac{7c}{m} + \frac{8}{n}) \leq n(1+\epsilon),$$

ha m és n olyan nagyok, hogy $\frac{1}{m} + \frac{7c}{m} + \frac{8}{n} < \epsilon$. □

Hasonló érveléssel megmutatható, hogy ha $\liminf_{n\to\infty} \frac{T_M(n)}{n} = \infty$, akkor M' választható úgy, hogy tetszőleges $c > 0$-ra és elég nagy n-re $T_{M'}(n) \leq cT_M(n)$ teljesüljön.

A szimulációs eredményeket azon az áron értük el, hogy mind a jelek számát, mind a belső állapotok számát megnöveltük. Számítógépes hasonlattal élve: komolyabb, bonyolultabb „hardvert" használtunk. Nagyobb (több állapotú) a központi egység, és nagyobb a jelhalmaz is. A fordított irányt illetően megjegyezzük, hogy – konstansszoros hatékonyságvesztés árán – a szalagjelek száma a szokásos bináris kódolással 2-re csökkenthető.

Megemlítjük még, hogy a legutóbbi tétel huncutság abban az értelemben, hogy a szalagjelek számának korlátlan növelhetőségére építettük a bizonyítást. Az érdekes tanulság ebből az, hogy a Turing-gépmodellben a feladatok időigénye függ a jelkészlet méretétől.

7.4. A kiszámíthatóság alapfogalmai

Ebben és a következő néhány részben az algoritmussal való kiszámíthatóság határait feszegetjük. Hogy erről értelmesen beszélni tudjunk, pontosan meg kell határoznunk az algoritmus fogalmát. Azt fogjuk algoritmikusan kiszámíthatónak tekinteni, ami Turing-géppel kiszámítható. A továbbiakban legyen I egy nem üres véges halmaz. Olyan gépekkel fogunk foglalkozni, melyeknek az input *abc*-je I.

Definíció (rekurzíve felsorolható nyelv): *Az $L \subseteq I^*$ nyelvet* rekurzíve felsorolható*nak nevezzük, ha van olyan M Turing-gép, melyre $L = L_M$, azaz a gép által felismert nyelv éppen L.*

Ahogy már korábban említettük, ez annyit tesz, hogy van egy fél algoritmusunk az L (illetve a neki megfelelő igen-nem kérdés) eldöntésére. Megeshet ugyanis, hogy egy $s \in I^* \setminus L$ szóval elindítva az M nem áll meg. A végtelen ciklusok kizárásával kapjuk a következő fogalmat:

Definíció (rekurzív nyelv): *Az $L \subseteq I^*$ nyelv* rekurzív*, ha van olyan M Turing-gép, melyre $L = L_M$, és M minden $s \in I^*$ szóra megáll.*

Nyilvánvaló, hogy egy rekurzív nyelv rekurzíve felsorolható is. A rekurzív, illetve rekurzíve felsorolható nyelvek halmazára az $\mathcal{R}$, illetve $\mathcal{RE}$ jelölést használjuk:

$$\mathcal{RE} = \{L \subseteq I^* : \; L \text{ rekurzíve felsorolható}\}.$$

$$\mathcal{R} = \{L \subseteq I^* : \; L \text{ rekurzív}\}.$$

Analóg fogalmak értelmezhetők (parciális) függvényekre:

Definíció (parciálisan rekurzív függvény): *Az $f : I^* \to I^*$ parciális függvény* parciálisan rekurzív *függvény, ha létezik olyan M Turing-gép, hogy $f = f_M$. Ha ezen túl még f minden $s \in I^*$ inputra értelmezve van, akkor f egy* rekurzív *függvény.*

A meghatározásból azonnal kiviláglik, hogy egy rekurzív függvény egyben parciálisan rekurzív is. A *rekurzív* elnevezés onnan ered, hogy a rekurzív függvények éppen azok a függvények, melyek bizonyos egyszerű függvényekből rekurzió segítségével felépíthetők. Ezzel a szintetikusnak mondható megközelítéssel nem foglalkozunk. A *rekurzíve felsorolható* kifejezés arra szándékozik utalni, hogy a nyelv szavai egy alkalmas eljárással felsorolhatók. Erre később (7.7.2. szakasz) még visszatérünk.

A rekurzív nyelveket és a rekurzív függvényeket fogjuk algoritmussal kezelhető nyelveknek és függvényeknek tekinteni. Felmerülhet a kérdés, hogy nem túl szűk-e ez a meghatározás. Nincsenek-e olyan algoritmusok, amelyek nem valósíthatók meg Turing-géppel? Például Pascal-programmal nem lehet-e több függvényt kiszámítani? Első hallásra talán meglepő lesz a válasz: számos kísérlet ellenére sem sikerült eddig olyan algoritmusfogalmat megadni, amely egyrészt megvalósítható a gyakorlatban, másrészt a Turing-kiszámíthatóságnál erősebb. Így például, ami elvégezhető Pascal-programmal, arra szerkeszthető Turing-gép is. Erre – a közvetlen elérésű gép kapcsán – látunk majd bizonyítékot. A Turing-gépeknek ezt a figyelemreméltó tulajdonságát fogalmazza meg a *Church–Turing-tézis*.

Church–Turing-tézis: *Ami algoritmussal – azaz véges eljárással – kiszámítható (eldönthető), az Turing értelmében kiszámítható (eldönthető). Nevezetesen:*

- *Egy $f : I^* \to I^*$ parciális függvény kiszámítható $\Leftrightarrow$ f parciálisan rekurzív.*
- *Egy $f : I^* \to I^*$ (teljes) függvény kiszámítható $\Leftrightarrow$ f rekurzív.*
- *Egy $L \subseteq I^*$ nyelvre a nyelvbe tartozás problémája algoritmussal eldönthető $\Leftrightarrow$ L rekurzív.*

Hangsúlyoznunk kell, hogy ez a tézis nem tekinthető formálisan bizonyított állításnak. Azt a *tapasztalati tényt* fejezi ki csupán, hogy eddig nem sikerült a Turing-gépek erejét meghaladó realisztikus számítási modellt találni. Megtörténhet – noha nehéz elképzelni –, hogy valaki kitalál egy megvalósítható számítóeszközt, amivel nem rekurzív függvények is kiszámíthatók. Egy ilyen fejlemény megdöntené a Church–Turing-tézist.

A Church–Turing-tézist elfogadva a következő két állítás úgy értelmezhető, hogy nem lehet mindent algoritmussal eldönteni, illetve kiszámítani. Vannak algoritmussal nem felismerhető nyelvek és nem kiszámítható függvények.

Állítás: *Van olyan $L' \subseteq I^*$ nyelv, amely nem rekurzíve felsorolható.*

Bizonyítás: Egy Turing-gép leírható véges jelsorozattal, pl. magyarul. Ezért az összes gép számossága megszámlálható. Ez annyit tesz, hogy mind felsorolható természetes számokkal sorszámozva. Legyen $M_0, M_1, M_2, \ldots$ egy ilyen felsorolás. A gép egyértelműen meghatározza az általa felismert nyelvet, vagyis a rekurzíve felsorolható nyelvek is sorszámozhatók a természetes számokkal úgy, hogy egy se maradjon ki: $L_{M_0}, L_{M_1}, L_{M_2}, \ldots$. Elég megmutatni ezután, hogy az $L \subseteq I^*$ nyelveket nem lehet ilyen módon megszámozni. Ebből egyből következik, hogy van olyan L' nyelv, ami nem nyelve egyetlen gépnek sem. (A halmazelmélet elemeiben jártas olvasónak: a rekurzíve felsorolható nyelvek halmaza megszámlálható, az összes nyelvé pedig kontinuum számosságú.)

Az I^* elemei, a véges hosszúságú, I-beli jelekből képzett szavak is megszámozhatók természetes számokkal. Hasznos számozást ad a *kanonikus felsorolás*: vegyük először az üres szót, majd soroljuk fel az 1 hosszúakat, utánuk a 2 hosszúakat, és így tovább. Az azonos hosszúságú szavak az I^* lexikografikus rendezése szerint kövessék egymást. Ehhez feltesszük, hogy adott az I halmazon egy rendezés. A kanonikus felsorolás az I^* szavainak egy $w_0, w_1, w_2, \ldots$ sorrendjét adja.

Megmutatjuk ezután, hogy van olyan $L' \subseteq I^*$ nyelv, amely nem lehet benne a rekurzíve felsorolható nyelvek $L_{M_0}, L_{M_1}, L_{M_2}, \ldots$ sorozatában. Az L' nyelvnek a w_i szó pontosan akkor legyen eleme, ha $w_i \notin L_{M_i}$, $i = 1, 2, \ldots$. Az L' definíciója korrekt. Ugyanis tetszőleges $w \in I^*$ szóról egyértelműen rendelkeztünk, mert minden szó pontosan egyszer szerepel a kanonikus felsorolásban.

Az L' nyelv nem egyezhet meg egyetlen L_{M_i} alakú nyelvvel sem, hiszen a w_i szó L' és L_{M_i} közül az egyiknek eleme, a másiknak nem. Az L' tehát nem rekurzíve felsorolható. □

Az eljárást, amellyel L'-t meghatároztuk, *Cantor-féle átlós módszer*nek nevezzük. Az elnevezés egyik lehetséges magyarázata a következő. Az I feletti nyelvek és a végtelen „igen–nem" sorozatok között I^* kanonikus felsorolása alapján kölcsönösen egyértelmű megfeleltetés hozható létre: egy $L \subseteq I^*$ nyelvnek az a sorozat felel meg, melynek az i-edik tagja pontosan akkor „igen", ha L-ben benne van a w_i szó. Készítsünk egy táblázatot, aminek az i-edik sora az L_{M_i}-nek megfelelő igen-nem sorozat. A táblázat oszlopait a w_i szavakkal indexeljük.

	w_0	w_1	...	w_i	w_{i+1}	...
L_{M_0}	**nem**	nem	...	nem	nem	...
L_{M_1}	igen	**nem**	...	nem	nem	...
$\vdots$						
L_{M_i}	nem	igen	...	**igen**	nem	...
$L_{M_{i+1}}$	igen	nem	...	nem	**nem**	...
$\vdots$						
L'	**igen**	**igen**	...	**nem**	**igen**	...

Az L' nyelv sorozatát – ami a rajzon az alsó sorban van – úgy kapjuk , hogy a táblázat főátlójában levő elemeket az ellenkezőjükre változtatjuk. A Cantor-módszer alapvető eszköz a kiszámíthatatlansági eredmények igazolásában.

Függvényekre a megfelelő állítás az előzőhöz hasonlóan bizonyítható. Itt azt lehet megmutatni, hogy az összes $f : I^* \to I^*$ függvényt nem lehet felsorolni.

Állítás: *Létezik olyan $f : I^* \to I^*$ parciális függvény, amely nem parciálisan rekurzív.* □

7.5. Az univerzális Turing-gép

Ennek az előkészítő jellegű résznek a tanulsága úgy foglalható össze, hogy *fordítóprogramok pedig léteznek*. Ezen senki sem lepődik meg különösebben, hiszen se szeri, se száma a különféle fordítóprogramoknak. Olyan programok ezek, amelyek képesek más programokat értelmezni, futtatni. Szinte el sem lehet képzelni nélkülük a számítógépes világot. Létezésük tehát hasznos tény. Ugyanakkor, mint később látni fogjuk, természetesen vezetnek algoritmussal kezelhetetlen feladatokhoz.

Az univerzális Turing-gépek a fordítóprogramok népes családján belül az interpreterek közé tartoznak. Bemenetük két részből áll. Az egyik komponens az interpretálandó program, ami egy M Turing-gép *leírása*. Az M-ről kényelmi okokból feltesszük, hogy egy szalagja van, a bemenő *abc*-je $I = \{0, 1\}$, és egyetlen elfogadó állapota van ($|F| = 1$). A szimulációk kapcsán láttuk, hogy az elfogadó/kiszámító képesség szempontjából ezek nem lényeges megszorítások: bármely N gép átalakítható ilyenné úgy, hogy a kapott új gép is az L_N nyelvet ismerje fel, illetve az f_N függvényt számolja ki.

Az univerzális gép bemenetének másik része az interpretálandó M gép tetszőleges $s \in I^*$ bemenete. Az univerzális gép az M leírását, más szóval *kódját* értelmezve lépésről lépésre utánozza M működését az s bemeneten.

Az első teendő tehát, hogy az interpretálandó M gépből „adatot" csináljunk, amit majd az univerzális gép olvasni tud. Ez nem különösebben nehéz. Arra kell ügyelnünk csupán, hogy minden M gép kódja egy véges bitsorozat legyen, és a kódolás/dekódolás algoritmussal (Turing-géppel vagy Pascal programmal) elvégezhető legyen.

A Turing-gépek egy lehetséges kódolása

Tegyük fel, hogy $k = 1$, $M = (Q, T, I, ü, \delta, q_0, F)$, $I = \{0, 1\}$ és $|F| = 1$. Azt is feltehetjük, hogy a gép megadásában szereplő szimbólumok mind számok. Mondjuk legyen

- $I = \{0, 1\}$, $T = \{0, 1, \dots, t\}$, $ü = t$,
- $Q = \{0, 1, \dots, q\}$, $q_0 = 0$, $F = \{q\}$,
- balra $= 0$, jobbra $= 1$, helyben $= 2$

Ekkor az M Turing-gép leírása, kódja legyen:

$$q\#t\#q_1\#x_1\#q_1'\#x_1'\#m_1'\# \dots \#q_r\#x_r\#q_r'\#x_r'\#m_r'\#\#,$$

ahol a megfelelő számokat binárisan írjuk le, továbbá δ értékeit (mindazokon a helyeken, ahol δ értelmezett) a következőképpen soroljuk fel:

$$\text{a } \delta(q_i, x_i) = (q_i', x_i', m_i') \text{ tény kódja } q_i\#x_i\#q_i'\#x_i'\#m_i'.$$

Itt q_i, q_i' állapotok, x_i, x_i' szalagjelek kódjai, m_i' pedig fejmozgást jelöl. Feltehető az is (pl. az $\# = 11$, $0 = 00$, $1 = 01$ átírással), hogy M leírása egy I^*-beli ($I = \{0, 1\}$) szó. A kódolás lényeges vonásait így összegezhetjük:

Minden szóba jövő M gépet egy $w \in I^*$ szóval írunk le. Tetszőleges $w \in I^*$ szóhoz legfeljebb egy gép van – ennek jele legyen M_w –, amelynek a kódja w. Az M ismeretében a w kód algoritmussal kiszámítható. A w leírás ismeretében pedig az M_w gép jellemzői (állapotai, átmenetfüggvénye, stb.) algoritmussal megkaphatók.

A következő tételben pontosan megfogalmazzuk, hogy mit értünk univerzális Turing-gép alatt, és vázlatosan meg is adunk egyet. Figyelemre méltó a konstrukció analógiája a számítógépes programozási gyakorlatból ismert interpreterekkel (mint pl. a BASIC nyelv interpreterei).

Tétel: *Van olyan 3-szalagos U Turing-gép, amelyre teljesül a következő: ha $w, s \in I^*$, és M_w létezik, akkor az U gép a $w\#s$ bemenetet pontosan akkor fogadja el (utasítja el, kerül vele végtelen ciklusba), ha M_w az s bemenetet elfogadja (elutasítja, végtelen ciklusba kerül vele).*

Bizonyítás: Csak vázoljuk az U gép szerkezetét. A nem különösebben érdekes részleteket mellőzzük. Az U első szalagja a $w\#s$ inputot tartalmazza, és csupán a w értelmezgetésére (lényegében az M_w átmenetfüggvényét leíró táblázatnak a böngészésére) szolgál. Az U gép második szalagja M_w egyetlen szalagjának felel meg. Tartalma és a fej helyzete az U működése során mindig azonos az M_w éppen szimulált lépésekor tapasztalható szalagtartalommal, illetve fejpozícióval. U harmadik szalagja az M_w-nek a szimulált helyzetben érvényes belső állapotát (pontosabban annak leírását) tartalmazza.

U a szimuláció tényleges megkezdése előtt egy kis előkészítő munkát végez. Először ellenőrzi, hogy M_w létezik-e. Ha nem, akkor itt megáll elutasító állapotban. Ellenkező esetben átmásolja az s inputot a második szalagjára, és a harmadik szalagra a kezdőállapot kódját jegyzi fel. Az előkészítő munka után U szalagjai így festenek:

$w\#s$
s
q_0

Az U az M_w gép egy lépését több lépésben szimulálja a következők szerint: a w leírás alapján meghatározza, hogy M_w mit lépne, ha U második szalagján a fej alatt található jelet a harmadik szalagon ábrázolt belső állapotban látja. Ha mondjuk a szalagjel a, az állapot pedig q, akkor w-ből a $\delta(q, a)$ leírását kell értelmeznie. Ennek ismeretében meglépi M_w lépését; ennek megfelelően módosítja a második és harmadik szalagot. Könnyen látható, hogy egy ilyen szimulációs lépés tényleg megvalósítható Turing-géppel. U szalagjai közvetlenül az M_w gép i-edik lépésének szimulációja után így néznek ki:

$w\#s$
M_w szalagja az i-edik lépés után
M_w belső állapota az i-edik lépés után

U akkor áll meg, ha M_w megáll. Ez esetben pontosan akkor fogadja el a $w\#s$ bemenetét, ha a megállás után az M_w elfogadó állapotának kódja van az utolsó szalagon. Mindezekből könnyen látható, hogy az itt körvonalazott U megfelel a tétel követelményeinek. □

7.6. Alapvető kiszámíthatatlansági tételek

Ha nem vagyok itt, a Hörpentőben keressen. Ha nem vagyok a Hörpentőben, akkor itt vagyok. Kivétel Violin, a fülrepesztő zenész. Ő, ha itt vagyok, keressen a Hörpentőben, ha meg a Hörpentőben vagyok, itt keressen. LÁZÁR ERVIN

(A Sróf mester ajtaján levő felirat *Berzsián és Dideki* történetéből.)

Most már megvannak az eszközeink, amelyekkel a kiszámíthatóság korlátaira vonatkozó első klasszikus eredményeket igazolni tudjuk. Megmutatjuk, hogy nem minden nyelv rekurzíve felsorolható; továbbá, hogy nem minden rekurzíve felsorolható nyelv rekurzív. Érvényesek tehát a következő szigorú tartalmazási relációk (2^{I^*} jelöli az összes I feletti nyelvből álló halmazt):

$$\mathcal{R} \subset \mathcal{RE} \subset 2^{I^*}.$$

A nyelvek definíciójában továbbra is az előző részben szereplő Turing-gépekre fogunk szorítkozni. Ezek egyszalagosak, az input *abc*-jük $\{0, 1\}$, és egyetlen elfogadó állapotuk van. Építeni fogunk az univerzális gép tárgyalásakor ismertetett kódolásra is.

7.6.1. A diagonális nyelv – egy nem rekurzíve felsorolható nyelv

Mint már láttuk, létezik nem rekurzíve felsorolható nyelv. Egy ilyen L' nyelv létezését a Cantor-féle diagonális eljárással mutattuk ki. Itt egy másik ilyen konstrukciót ismertetünk, amit Cantor ötletének közvetlenebb alkalmazásával nyerünk.

Tekintsük azon Turing-gépeket, amelyek *nem fogadják el a saját kódjukat*. Tegyük fel, hogy az ezen gépek kódjaiból álló L_d nyelv rekurzíve felsorolható, azaz létezik olyan M Turing-gép, amely éppen ezt a nyelvet ismeri fel. A felismert nyelv fogalmából tudjuk, hogy M akkor és csak akkor ismeri fel egy N gép kódját, ha N kódja benne van a nyelvben. Másrészt a nyelvet éppen úgy adtuk meg, hogy az N kódja pontosan akkor van benne, ha N nem ismeri fel a saját kódját. A két állítást az $M = N$ speciális esetre összerakva azt kapnánk, hogy M akkor és csak akkor ismeri fel a saját kódját, ha nem ismeri fel a saját kódját, ami képtelenség. Tehát az L_d nyelv nem lehet rekurzíve felsorolható. Nézzük ezt egy kicsit pontosabban! Az L_d *diagonális nyelv* a következő:

$$L_d = \{w \in I^*;\ \text{az } M_w \text{ gép létezik, és } w \notin L_{M_w}\}.$$

Tétel: L_d *nem rekurzíve felsorolható.*

Bizonyítás: Indirekt érvelve tegyük fel, hogy a diagonális nyelv rekurzíve felsorolható. Ez annyit tesz, hogy van olyan M Turing-gép, amelyre $L_d = L_M$. Nézzük most ennek az M gépnek a $w \in I^*$ kódját, másként mondva azt a szót, amellyel $M = M_w$ fennáll. Az L_d nyelv és a w szó viszonyát illetően két lehetőség van: vagy $w \in L_d$, vagy pedig $w \notin L_d$. Megmutatjuk, hogy mindkét feltevés ellentmondáshoz vezet, igazolva ezzel a tételt.

Ha $w \in L_d$, akkor L_d definíciója szerint $w \notin L_{M_w}$. De az utóbbi nyelv éppen L_d, tehát $w \notin L_d$, ami ellentmond kiinduló feltevésünknek.

Ha pedig abból indulunk ki, hogy $w \notin L_d$, akkor L_d definíciója szerint $w \in L_{M_w}$, hiszen M_w létezik. Újfent használva az $L_d = L_{M_w}$ egyenlőséget azt kapjuk, hogy $w \in L_d$, ami ismét képtelenség. A bizonyítás ezzel teljes. □

7.6.2. Az univerzális nyelv – egy rekurzíve felsorolható, de nem rekurzív nyelv

A következő algoritmikus kérdés is gépekkel foglalkozik: adott egy M gép és egy $s \in I^*$ szó; döntsük el, hogy M elfogadja-e s-et. Első megoldási kísérletünk az lehetne, hogy futtatjuk M-et, és megnézzük, mi történik. Ez megy is akkor, ha a kérdésre a válasz igenlő, azaz $s \in L_M$. Véges sok lépés után megtudjuk a választ. Ha viszont s végtelen ciklust eredményez, akkor bajban vagyunk. A gép csak megy és megy, és mi nem tudjuk, hogy tényleg végtelen ciklusban van-e, vagy csak egyszerűen egy sokáig tartó véges számítással van dolgunk.

Alaposabban megnézve a kérdést arra jutunk, hogy az algoritmusnak, amit keresünk, valamiféle szuperalgoritmusnak kellene lennie. Olyannak, ami minden gépnél okosabb, hiszen mindent tud, amit a többiek. Mindjárt kiderül, hogy ilyen módszer nincs. A problémának – ami egy eldöntendő kérdés – megfelelő nyelv L_u, az *univerzális nyelv*:

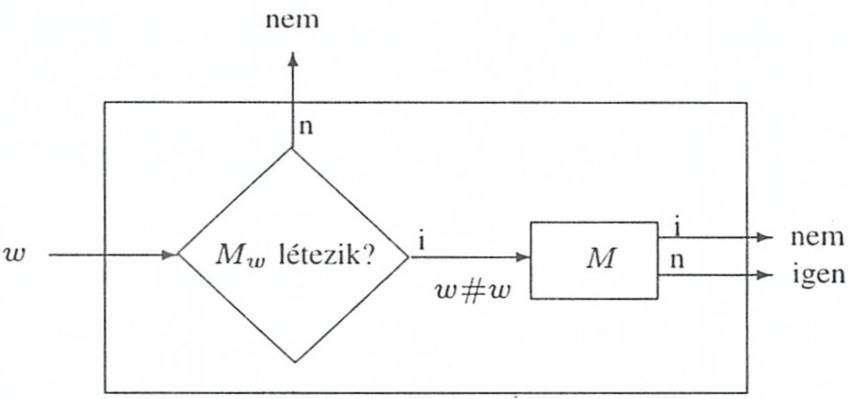
$$L_u = \{w\#s \in I^*;\ \text{ az } M_w \text{ gép létezik, és } s \in L_{M_w}\}.$$

Az univerzális nyelv tehát azon *(kód,input szó)* párokból áll, melyekre a kóddal leírt Turing-gép felismeri az illető input szót. Az elnevezés azon az egyszerű észrevételen alapszik, hogy ez a nyelv éppen *az univerzális Turing-gépek nyelve*: pontosan azokból a szavakból áll, amelyeket egy univerzális gép elfogad. Tömören ezt úgy írhatjuk, hogy $L_u = L_U$, ahol U tetszőleges univerzális gép. Innen azonnal látjuk, hogy L_u rekurzíve felsorolható.

Meg fogjuk mutatni másfelől, hogy L_u nem rekurzív. Ha volna egy mindig megálló M algoritmus, ami L_u-t felismerné, akkor szerkeszteni tudnánk egy másik – az M eljárást „hívó" – módszert, ami felismerné a diagonális nyelvet. Ez pedig lehetetlen, hiszen L_d-ről láttuk, hogy nem rekurzíve felsorolható. Érvényes tehát a következő:

Tétel (A. Turing, 1936): *L_u egy rekurzíve felsorolható, de nem rekurzív nyelv.*

Bizonyítás: L_u-t éppen az univerzális Turing-gépek ismerik fel, tehát L_u rekurzíve felsorolható.
Tegyük fel ezután indirekte, hogy L_u rekurzív; legyen M egy mindig megálló Turing-gép, ami felismeri L_u-t. Legyen M' az a Turing-gép, amelynek működését az alábbi folyamatábra mutatja:



Az M' gép a $w \in I^*$ bemenettel elindítva a következő fő lépéseket végzi:
(1) Ellenőrzi, hogy M_w létezik-e. Ha nem, akkor elutasító állapotban megáll.
(2) Ellenkező esetben, ha M_w létezik, akkor elindítja M-et a $w\#w$ inputtal. Ha M elfogadó állapotban áll meg, akkor M' elutasító állapotban végez, ha M elutasító állapotban állt meg, akkor elfogadja a w bemenetet.

Az M' gép mindig megáll, mert az (1) lépés tesztje algoritmussal megvalósítható, és M mindig megáll. Világos továbbá, hogy M' pontosan akkor fogadja el w-t, ha M_w létezik és $w \notin L_{M_w}$. Más szóval M' éppen az L_d nyelvet fogadja el. Ez pedig képtelenség, hiszen L_d nem rekurzíve felsorolható, és ezért nem lehet egyetlen gép nyelve sem. M létezését feltételezve ellentmondást kaptunk, igazolva ezzel, hogy L_u nem rekurzív. □

7.7. Összefüggések a kiszámíthatósági fogalmak között

Az eddig vett kiszámíthatósági fogalmak közötti viszonyokkal foglalkozunk a következőkben. Először a rekurzíve felsorolható és a rekurzív nyelvek közötti kapcsolatot vesszük szemügyre, majd a függvények számításának és a nyelvek felismerésének feladatait vetjük össze.

7.7.1. Rekurzivitás és rekurzíve felsorolhatóság

A nyelvfelismerési feladatokról megállapítottuk, hogy igen-nem problémáknak felelnek meg. Valamely P tulajdonsággal rendelkező szavak halmazát kell felismerni. A nyelvbe azok a szavak tartoznak, melyekre P igaz. Érdekes szerepe van itt a P tulajdonság $\neg P$ tagadásának. A $\neg P$ pontosan akkor teljesül egy $s \in I^*$ szóra, ha P nem teljesül s-re. Ezért, ha a P igenpéldányaiból álló nyelv L, akkor a $\neg P$ igenpéldányaiból álló nyelv az L *komplementere*: $I^* \setminus L$. Ha van egy mindig megálló M algoritmusunk a P eldöntésére, azaz L felismerésére, akkor a $\neg P$ tulajdonságot is el tudjuk dönteni. Csupán az M elfogadó és elutasító állapotait kell felcserélni. Az így kapott M' gép az $I^* \setminus L$ nyelvet ismeri fel.

Ha csak egy fél algoritmusunk van, azaz $L = L_M$, de M nem áll meg minden bemeneten, akkor az állapotok felcserélésével nem érünk sokat. Az olyan $s \in I^*$ bemenetekre, amelyekre M nem áll meg, sosem tudjuk meg a választ véges sok lépés után. Ha viszont a P és a $\neg P$ eldöntésére is van egy-egy fél algoritmusunk, akkor ezek összerakhatók egy egésszé. A két gépet párhuzamosan működtetve véges időben megkapjuk a választ minden bemenetre. Ezeket az észrevételeket pontosan is megfogalmazzuk. Bevezetünk előbb egy fogalmat, amit a bonyolultságelméleti fejezetben is használni fogunk.

A nyelvekből álló halmazokat (2^{I^*} részhalmazait) *nyelvosztályoknak* nevezzük. Nyelvosztályokra eddigi példáink $\mathcal{R}$, $\mathcal{RE}$, 2^{I^*}. Legyen $X \subseteq 2^{I^*}$ egy nyelvosztály. Ekkor a *komplementer nyelvosztály*, coX az X-beli nyelvek komplementereiből áll:

$$\text{co}X = \{L \subseteq I^* : \quad I^* \setminus L \in X\}.$$

Az így értelmezett komplementerképzés fontos sajátja, hogy megőrzi a nyelvosztályok közötti tartalmazási viszonyokat:

$$X \subseteq Y \subseteq 2^{I^*} \implies \text{co}X \subseteq \text{co}Y.$$

Ugyanis ha $L \in \text{co}X$, akkor definíció szerint $I^* \setminus L \in X$. Az $X \subseteq Y$ tartalmazás miatt $I^* \setminus L \in Y$, amiből pedig $L \in \text{co}Y$. Szintén egyszerűen igazolható a tetszőleges X nyelvosztályra érvényes $\text{co}(\text{co}X) = X$ összefüggés.

A bevezetett jelöléssel a következőképpen fogalmazhatjuk meg észrevételeinket:

Tétel:
(1) $\mathcal{R} = co\mathcal{R}$
(2) $\mathcal{R} = \mathcal{RE} \cap co\mathcal{RE}$

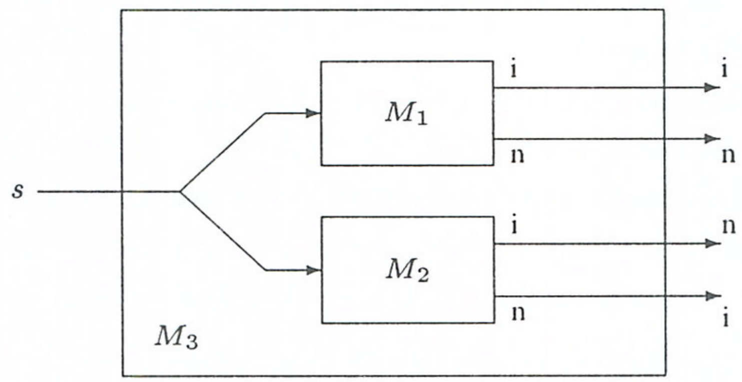
Bizonyítás: (1) Tegyük fel, hogy $L \in \mathcal{R}$. Legyen ennek megfelelően M egy Turing-gép, mely minden inputon megáll és az L nyelvet fogadja el. Az M elfo-

gadó és elutasítva megálló állapotainak felcserélésével olyan gépet nyerünk, amelyik éppen az L komplementerét, az $I^* \setminus L$ nyelvet ismeri fel. Nyilván az új gép is mindig megáll, tehát $I^* \setminus L$ is rekurzív. Beláttuk ezzel, hogy $\mathrm{co}\mathcal{R} \subseteq \mathcal{R}$. A fordított irányú tartalmazás innen már adódik a co-operátorra megismert szabályokból:

$$\mathcal{R} = \mathrm{co}(\mathrm{co}\mathcal{R}) \subseteq \mathrm{co}\mathcal{R}.$$

(2) A nyilvánvaló $\mathcal{R} \subseteq \mathcal{RE}$ egyenlőtlenségből komplementerképzéssel nyerjük, hogy $\mathcal{R} = \mathrm{co}\mathcal{R} \subseteq \mathrm{co}\mathcal{RE}$, így összességében $\mathcal{R} \subseteq \mathcal{RE} \cap \mathrm{co}\mathcal{RE}$. A fordított irányú tartalmazás igazolásához tegyük fel, hogy $L \in \mathcal{RE} \cap \mathrm{co}\mathcal{RE}$. Legyen ennek megfelelően M_1, illetve M_2 két Turing-gép, melyek az L, illetve az $I^* \setminus L$ nyelvet ismerik fel. Ezekből egy mindig megálló M_3 algoritmus szerkeszthető, melye $L = L_{M_3}$.

Legyen $s \in I^*$ a bemenő szó. Az M_3 gépet szemléletesen úgy képzelhetjük el, hogy az M_1 és M_2 gépek egymás mellett, egymástól függetlenül párhuzamosan dolgoznak ugyanazzal az s inputtal:



Az M_3 pontosan akkor álljon meg, ha M_1 és M_2 valamelyike megáll. M_3 akkor fogad el, ha a megállás M_1 elfogadó, vagy pedig M_2 elutasító állapotában történt. Világos, hogy az M_3 az L nyelvet ismeri fel. Az is igaz, hogy mindig megáll, hiszen ha $s \in L$, akkor M_1, ha pedig $s \notin L$, akkor M_2 fog biztosan megállni véges sok lépés után.

M_3 megvalósítható párhuzamosság nélkül is. A megfelelő Turing-gép szalagjai egy részén végzi az M_1 munkáját, egy ettől diszjunkt másik részén pedig az M_2 teendőit. M_3 felváltva szimulálja a két gép lépéseit: a $2i-1$-edik lépésben az M_1 i-edik, a $2i$-edik lépésben pedig az M_2 i-edik lépését teszi meg. Ezáltal, ha valamelyik gép megáll a k-adik lépésében, akkor M_3 is megáll, legkésőbb a $2k$-adik lépésben.

Beláttuk tehát, hogy L rekurzív nyelv, vagyis $\mathcal{R} \supseteq \mathcal{RE} \cap \mathrm{co}\mathcal{RE}$. A másik tartalmazással együtt ez bizonyítja a (2) állítást.□

Feladat: Az előző érvelésben az M_1 és M_2 párhuzamos futását úgy alakítottuk sorossá, hogy felváltva léptettük őket. Milyen más módokon fésülhetők össze a két gép lépései úgy, hogy a bizonyítás továbbra is működjön?

A tételt olyan állításnak tekinthetjük, amely a nyelvosztályainknak a komplementer műveletre való zártságával foglalkozik. Kiderül, hogy a rekurzív nyelvek összessége zárt a komplementer képzésére – azaz $L \in \mathcal{R}$ esetén $I^* \setminus L \in \mathcal{R}$ is igaz –, míg a rekurzíve felsorolható nyelvekre ez nem teljesül. A következő feladat az unió- és a metszetképzés műveleteire való zártságról szól.

Feladat: Tegyük fel, hogy az L_1 és L_2 nyelvek rekurzíve felsorolhatók (rekurzívek). Igazoljuk, hogy $L_1 \cap L_2$ és $L_1 \cup L_2$ is rekurzíve felsorolható (rekurzív). (Az előző tételben látottakhoz hasonlóan működtessük párhuzamosan az L_1 és L_2 nyelveket felismerő M_1 és M_2 gépeket.)

7.7.2. Függvények és halmazok (nyelvek)

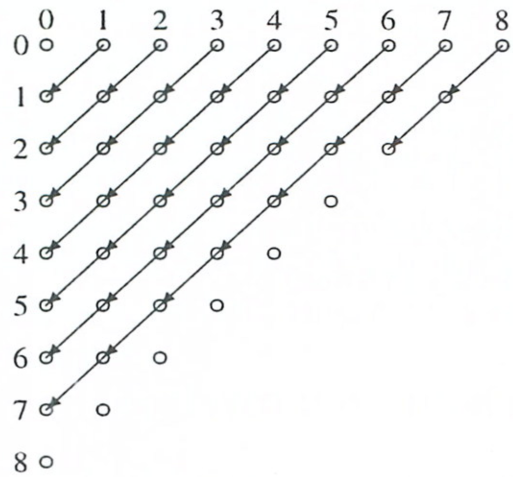
A Turing-gépeknek kétféle működési módjával ismerkedtünk meg. Az egyik a nyelvek felismerése, a másik pedig a függvények számítása. Az előbbinél a válasz egyetlen bittel ábrázolható (elfogadás vagy annak a hiánya), a másodiknál hosszabb is lehet az eredmény. Ebben az értelemben a függvények számítása általánosabb feladat, mint a nyelvek felismerése. Az általános és a speciális közötti összefüggések szinte mindig érdekesek. Itt a kiszámíthatóság szempontjából vetjük össze a két fogalmat. Később – a kisebb bonyolultságú feladatok tanulmányozásakor – újra előkerül majd a kétféle működés közötti viszony.

A kiszámíthatóság nézőpontjából a rekurzíve felsorolható nyelvek a parciálisan rekurzív függvényekkel, míg a rekurzív nyelvek a rekurzív függvényekkel hozhatók természetes kapcsolatba. Ezeket fogalmazza meg a következő két tétel.

Tétel: *Az $L \subseteq I^*$ nyelv akkor és csak akkor rekurzíve felsorolható, ha van olyan $f : I^* \rightarrow I^*$ parciálisan rekurzív függvény, melynek értékkészlete éppen az L nyelv (szokásos jelöléssel $Im(f) = L$).*

Bizonyítás: $\Rightarrow$: Tegyük fel, hogy L rekurzíve felsorolható. Ennek megfelelően van olyan M Turing-gép, melyre $L = L_M$. Legyen ezután M' olyan Turing-gép, mely kezdetben az $s \in I^*$ bemenő szót felmásolja az output szalagjára, azután pedig lépésről lépésre szimulálja az M gépet. Az egyetlen kivétel, hogy ha M elutasító állapotban áll meg, akkor M' végtelen ciklusba esik. Az M' gép tehát kizárólag az $s \in L$ szavakra áll meg; megálláskor s lesz az output szalagon. Az M' által kiszámított $f_{M'}$ függvény értékkészlete ezért éppen L.

⇐: Tegyük fel, hogy f egy parciálisan rekurzív függvény, mondjuk $f = f_M$, ahol M egy Turing-gép. Tekintsük az I^* szavainak $w_0, \ldots, w_n \ldots$ kanonikus felsorolását (a rendezés hosszúság szerint növekvően, azonos hosszúak között lexikografikusan). Használni fogjuk még a természetes számokból álló párok egy felsorolását is. A következő ábrán látható táblázaton megyünk végig a délnyugati irányú átlók mentén. A sorozat eleje így alakul: $(0,0), (0,1), (1,0), (0,2), (1,1), (2,0)$, stb.



A felsorolások két tulajdonsága lesz fontos a számunkra. Egyrészt mindkettő megvalósítható algoritmussal. A sorozatok adott elemének az ismeretében a következő elem számítással meghatározható. A másik lényeges vonásuk, hogy mindkét felsorolás kimerítő; mindegyik szóra, illetve párra sor kerül valamikor.

Az M' Turing-gép a teendőit az (i,j) párok sorozatához kapcsolódóan, a sorozat mentén haladva végzi. Tegyük fel, hogy $s \in I^*$ az M' bemenete. Az (i,j) párhoz kapcsolódóan M' a következőket teszi: szimulálja az M első $\leq i$ lépését a w_j szón. Ha ezalatt M megáll és o outputot produkál, akkor ellenőrzi, hogy $s = o$ teljesül-e. Ha igen, akkor M' megáll és elfogadja s-et. Minden más esetben (M nem áll meg i lépésen belül, vagy megálláskor az eredmény nem s) a felsorolás szerint következő párral folytatja a munkát. Ha ez (i',j') akkor M első i' lépését teszi meg a $w_{j'}$ szóval, stb.

Mi lesz az M' gép $L_{M'}$ nyelve? Világos, hogy $L_{M'} \subseteq \mathrm{Im}(f)$, hiszen M' csak olyan s szót fogad el, ami megkapható mint M egy számításának az eredménye. Fordítva, tegyük fel, hogy $s \in \mathrm{Im}(f)$. Van tehát egy olyan j, hogy $s = f_M(w_j)$. Tegyük fel, hogy M a w_j bemeneten i lépésben kapja meg az s eredményt. Ekkor M' – ha előbb nem – az (i,j) pár feldolgozásakor elfogadja s-et. Ezzel igazoltuk az $L_{M'} \supseteq \mathrm{Im}(f)$ tartalmazást is. Igaz tehát, hogy $L_{M'} = \mathrm{Im}(f)$. □

A bizonyítás második felének ötletét felhasználva tetszőleges rekurzíve felsorolható L nyelvhez szerkeszthető olyan algoritmus, ami éppen az L nyelv szavait *sorolja fel* valamilyen sorrendben. Innen ered a *rekurzíve felsorolható* kifejezés.

A rekurzivitásra vonatkozó tétel kimondása előtt emlékeztetünk egy hasznos fogalomra. Az egyszerűség kedvéért feltesszük, hogy $0, 1 \in I$. Egy $L \subseteq I^*$ nyelv *karakterisztikus függvénye*, χ_L a következő:

$$\chi_L(s) = \begin{cases} 1 \in I^*, & \text{ha } s \in L \\ 0 \in I^*, & \text{ha } s \notin L \end{cases}$$

Tétel: *Az $L \subseteq I^*$ nyelv pontosan akkor rekurzív, ha χ_L egy rekurzív függvény.*

Bizonyítás: $\Rightarrow$: Tegyük fel, hogy L rekurzív: M olyan Turing-gép, mely L-et ismeri fel, és minden inputra megáll. Legyen ezután M' olyan Turing-gép, mely szimulálja M-et, majd 1-et vagy 0-t ír az output szalagjára, annak megfelelően, hogy M elfogadó vagy elutasító állapotban állt meg. Világos, hogy M' éppen az L karakterisztikus függvényét számolja ki.

$\Leftarrow$: Tegyük fel, hogy χ_L rekurzív függvény, és ennek megfelelően az M Turing-gép a χ_L függvényt számítja ki. Legyen M' egy olyan Turing-gép, mely lépésről lépésre szimulálja M-et; végül elfogadó, illetve elutasító állapotban áll meg aszerint, hogy M output szalagján 1, vagy 0 a végeredmény. Ez az M' gép pontosan az L nyelvet ismeri fel. □

7.8. További eldönthetetlen problémák

Visszatérünk az algoritmussal nem megoldható feladatokhoz. Az ember először hajlamos ezeket afféle patologikus eseteknek tekinteni, amelyek nem mondanak sokat a természetesen felmerülő problémák iránt érdeklődőknek. Szeretnénk néhány példával érzékeltetni, hogy ez távolról sem igaz: ilyen reménytelenül nehéz problémákkal a legkülönbözőbb területeken is találkozhatunk. Egy meghatározással kezdjük, amellyel nem új fogalmat, hanem inkább egy új elnevezést rögzítünk.

Definíció (eldönthetetlen nyelv): *Az $L \subseteq I^*$ nyelvet* eldönthetetlen nyelv*nek nevezzük, ha L nem rekurzív.*

A szóhasználat azt takarja, hogy az L nyelvbe tartozás/nem-tartozás problémáját nem lehet algoritmussal eldönteni: nincs olyan algoritmus, mely véges lépésben meg tudná mondani, hogy egy adott szó benne van-e a nyelvben. Turing tétele szerint az a probléma, hogy egy Turing-gép egy adott inputot elfogad-e, algoritmikusan eldönthetetlen. Itt további példákat adunk eldönthetetlen problémákra. Az igazán érdekesek azok lesznek, amelyek a Turing-gépek formalizmusától függetlenül, természetesen megfogalmazhatók.

7.8.1. A Megállási probléma (Halting problem)

Egy program tesztelésekor fontos kérdés, hogy nincs-e benne végtelen ciklus. Jó lenne hatékony módszert találni ennek az eldöntésére. Legyünk szerényebbek: nézzük csak azt ez egyszerűbb kérdést, hogy egy adott program egy rögzített bemenettel megáll-e véges sok lépés után. Még ez az ártatlannak tűnő kérdés is eldönthetetlen[2].

A *Megállási probléma* kérdése az, hogy egy (a kódjával adott) Turing-gép adott bemenettel egyáltalán megáll-e. A problémához tartozó L_h nyelv a következő:

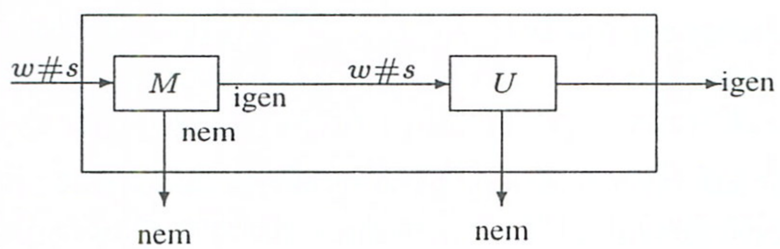
$$L_h = \left\{ w\#s \in I^* \;\middle|\; \begin{array}{l} \text{az } M_w \text{ gép létezik, és az } s \text{ bemenettel} \\ \text{elindítva véges sok lépésben megáll} \end{array} \right\}.$$

Az L_h nyelv nyilván tartalmazza az L_u univerzális nyelvet. A következő tétel gondolatmenete tulajdonképpen a két nyelv közötti algoritmikus kapcsolaton alapul.

Tétel: $L_h \in \mathcal{RE} \setminus \mathcal{R}$.

Bizonyítás: Lényegében az univerzális Turing-gép mutatja, hogy $L_h \in \mathcal{RE}$. Csak annyi módosítás szükséges, hogy megálláskor a gép *mindig* menjen át elfogadó állapotba. Az így megbütykölt gép pontosan akkor fogadja el a $w\#s$ bemenetet, ha ezzel U megáll, azaz ha $w\#s \in L_h$.

Az állítás fennmaradó részének igazolásához tegyük fel indirekte, hogy L_h rekurzív. Eszerint van olyan M Turing-gép, ami L_h-t ismeri fel, és mindig megáll. Legyen ekkor M' az a Turing-gép, mely a $w\#s$ inputon először az M gépet futtatja le. Ha M elutasítja a bemenetét, akkor M' álljon meg elutasító állapotban. Ha M elfogadó állapotban áll meg, akkor M' szimulálja az U univerzális Turing-gépet a $w\#s$ bemeneten, beleértve az elfogadást/elutasítást is.



[2] Itt komoly szerephez jut az a tény, hogy Turing-gépek szalagja végtelen. Ha – az igazi számítógépekhez hasonlóan – csak véges méretű tárat engedünk meg, akkor a feladat megoldhatóvá válik. A következő fejezetben a tár–idő-tételben adunk egy módszert a tárkorlátos végtelen ciklusok felismerésére. Ez azonban a tesztelés szemszögéből csak elvi jelentőségű; nem ismert gyakorlati szempontból elfogadható megoldás a feladatra.

Az M-re tett feltevés miatt M' mindig megáll. Nyilvánvaló ezen felül, hogy M' éppen az L_u univerzális nyelvet ismeri fel. Mindezek ellentmondanak annak, hogy L_u eldönthetetlen (Turing tétele). □

Valamivel több is mondható: a Megállási probléma akkor is algoritmikusan eldönthetetlen marad, ha az M_w gépet csupán az üres inputtal futtatjuk. Legyen

$$L_\epsilon = \{w \in I^* : \ M_w \text{ létezik és az } \epsilon \text{ (üres) inputon megáll}\}.$$

Tétel (A. Church, 1936):

$$L_\epsilon \in \mathcal{RE} \setminus \mathcal{R}.$$

Bizonyítás: Az L_ϵ nyelv rekurzíve felsorolhatósága így látható: a $w \in I^*$ inputból elkészítjük a $w\#\epsilon = w\#$ szót, majd ezzel a bemenettel lefuttatjuk az L_h megállási nyelvet felismerő gépet. Az így adódó algoritmus pontosan az L_ϵ nyelvet ismeri fel.

Az $L_\epsilon \notin \mathcal{R}$ állítás igazolására megmutatjuk, hogy az általános Megállási probléma visszavezethető erre a speciális esetre: egy M Turing-gépnek az s bemeneten való működése szimulálható egy olyan (az M-től és az s szótól egyaránt függő) Turing-géppel, amely üres input esetén felmásolja az s szót az input szalagjára, majd M működését lépésről lépésre utánozza. Ha tehát az L_ϵ nyelv eldönthető volna, akkor az általános Megállási feladat is eldönthető lenne. □

7.8.2. Hilbert 10. problémája

> *... ennek ősi és egyszerű íze van, mint talán az* Ezeregyéjszaká*nak...*
> JORGE LUIS BORGES*:* Ember a küszöbön

1900-ban Párizsban a világ matematikusainak kongresszusán David Hilbert egy azóta legendás hírűvé vált előadást tartott. Ebben a jövő fontos kutatási irányait próbálta meg kijelölni. Huszonhárom olyan kérdést fogalmazott meg, melyekről úgy vélte, hogy a megválaszolásuk lényeges előrelépést jelentene. Kifejezte azt a meggyőződését is, hogy a kérdések nehezek, és a XX. század nem lesz elég a megoldásukhoz. A két jóslat közül az utóbbi vált be kevésbé. A kérdések jó részét mára megoldották, és mindegyikkel kapcsolatban komoly eredmények születtek. A problémák jelentőségét illetően nem tévedett: óriási hatást gyakoroltak a tudomány fejlődésére. Nem kivétel ez alól a tizedik kérdés sem.

A tizedik problémában Hilbert célul tűzte ki, hogy adjunk általános módszert *diofantikus egyenletek* megoldhatóságának eldöntésére. A probléma pontos

megfogalmazásához szükségünk lesz egy fogalomra. *Egész együtthatós polinomon* egy olyan kifejezést értünk, amely változókból és egész számokból építhető fel az összeadás, kivonás és szorzás műveleteivel. Eszerint az $x^3y^2 - 5zu$ kifejezés tekinthető az x, y, z, u változók egy egész együtthatós polinomjának. Jelölje $\mathbb{Z}[x_1, \ldots, x_m]$ azon egész együtthatós polinomok összességét, melyekben a változók $x_1, x_2, \ldots, x_m$ közül kerülnek ki; például $x^3y^2 - 5zu \in \mathbb{Z}[x, y, z, u]$.

Tetszőleges $f(x_1, \ldots, x_m) \in \mathbb{Z}[x_1, \ldots, x_m]$ polinom felírható

$$f(x_1, \ldots, x_m) = \sum_{i_1=0}^{n_1} \cdots \sum_{i_m=0}^{n_m} a_{i_1 \ldots i_m} {x_1}^{i_1} \cdots {x_m}^{i_m}$$

alakban, ahol $n_1, \ldots, n_m \in \mathbb{Z}^+$ és az $a_{i_1 \ldots i_m}$ együtthatók pedig egész számok. Erről könnyen meggyőződhetünk, ha az f-et megadó kifejezésben felbontjuk a zárójeleket. Az f polinom *foka* az előző felírásban előforduló legnagyobb kitevőösszeg:

$$\deg f = \max\{i_1 + \ldots + i_m \mid a_{i_1 \ldots i_m} \neq 0\}.$$

Az

$$(*) \qquad f(x_1, \ldots, x_m) = 0$$

alakú egyenleteket *diofantikus egyenleteknek* nevezzük, ahol $f(x_1, \ldots, x_m)$ egész együtthatós polinom. A $(*)$ diofantikus egyenlet *megoldásán* egy olyan $(u_1, \ldots, u_m) \in \mathbb{Z}^m$ egészekből álló m-est értünk, melyre $f(u_1, \ldots, u_m) = 0$. A diofantikus egyenletet *megoldhatónak* nevezzük, ha van ilyen értelemben vett megoldása. A diofantikus egyenletek megoldhatóságának vizsgálata, megoldásaik keresése szinte a szám fogalmának létezése óta művelt, rendkívül gazdag terület. Ilyen egyenlet (pozitív) megoldásaiként adhatók meg például a *pitagoraszi számhármasok*: azok a természetes számokból álló (x, y, z) hármasok, melyek egy derékszögű háromszög oldalai lehetnek:

$$x^2 + y^2 - z^2 = 0.$$

Hilbert tizedik kérdése így fogalmazható: *adjunk egy olyan véges sok lépésből álló eljárást, mely tetszőleges egész együtthatós f polinomra eldönti, hogy az f-nek megfelelő $(*)$ egyenlet megoldható-e.*

Úgy tűnik, Hilbert a problémát abban a meggyőződésében tette közzé, hogy minden értelmesen megfogalmazott számítási problémára van algoritmus. Ezt a harmincas években Kurt Gödel, Alonzo Church és Alan Turing munkássága alaposan megcáfolta. Mint láttuk, vannak olyan feladatok, amelyekre nem létezik megoldó algoritmus. Ezek után már természetesen merült fel a lehetőség, hogy

a tizedik probléma feladata is egy ilyen kemény dió. Ebbe az irányba mutatott az a régen megfigyelt tény is, hogy a diofantikus egyenleteknek központi szerepük van a matematikai hátterű algoritmikus kérdésekben: sok fontos igen–nem problémát át tudtak fogalmazni diofantikus egyenlet megoldhatóságává.

A tizedik probléma sokáig ellenállt a kutatók ostromának, míg végül Martin Davis, Julia Robinson és mások munkáira építve 1970-ben Jurij Matijaszevics megmutatta, hogy *nem létezik olyan algoritmus, ami a* $(*)$ *alakú egyenletek megoldhatóságát eldöntené*. A tizedik problémában megfogalmazott algoritmikus kérdés eldönthetetlen.

A nevezetes eredmény ismertetése kényelmesebb lesz, ha nyelvek helyett természetes számok halmazaival foglalkozunk. A kanonikus felsorolás kölcsönösen egyértelmű megfeleltetést létesít $\{0,1\}^*$ szavai és a természetes számok között: az $s \in \{0,1\}^*$ szónak azon i index felel meg, melyre $s = w_i$. Ezzel egyszersmind kölcsönösen egyértelmű megfeleltetést kapunk a természetes számok részhalmazai és az $L \subseteq \{0,1\}^*$ nyelvek között. E megfeleltetés révén beszélhetünk arról, hogy egy $H \subseteq \mathbb{Z}^+$ számhalmaz rekurzív, illetve rekurzíve felsorolható. Ez egyszerűen azt jelenti, hogy a H-beli sorszámokkal azonosított szavak halmaza rekurzív, illetve rekurzíve felsorolható.

Legyen $m \geq 0$ és $f(x_1, x_2, \ldots, x_m, y) \in \mathbb{Z}[x_1, x_2, \ldots, x_m, y]$ egy egész együtthatós $m+1$-változós polinom és legyen

$$H_f = \left\{ v \in \mathbb{Z}^+ \;\middle|\; \begin{array}{l} \text{vannak olyan } u_1, u_2, \ldots, u_m \text{ természetes számok,} \\ \text{hogy } f(u_1, u_2, \ldots, u_m, v) = 0 \end{array} \right\}.$$

Definíció: *A* $H \subseteq \mathbb{Z}^+$ *halmaz* diofantikus halmaz, *ha van olyan* f *polinom, hogy* $H = H_f$.

A diofantikus halmaz fogalma mögött egy számítási modell húzódik meg, ami első látásra bizarrnak és szárnyaszegettnek tűnik. Ebben az f polinom jelenti a programot. A $v \in \mathbb{Z}^+$ számot a program akkor fogadja el, ha vannak olyan u_i természetes számok, hogy $f(u_1, u_2, \ldots, u_m, v) = 0$. Az rögvest látszik, hogy ez a furcsán értelmezett számítás megvalósítható Turing-géppel: az adott v bemenethez sorra vesszük az $(u_1, u_2, \ldots, u_m) \in (\mathbb{Z}^+)^m$ vektorokat; az $(u_1, u_2, \ldots, u_m)$ vektor kapcsán kiszámítjuk az $f(u_1, u_2, \ldots, u_m, v)$ helyettesítési értéket. Ha ez nulla, akkor elfogadjuk v-t, ha nem, akkor a következő vektorral próbálkozunk. A vektorok felsorolásával mint számítási feladattal már találkoztunk. Az $m = 2$ esetre adtunk egy módszert. Ez általánosítható tetszőleges m-re: sorra vesszük azokat a vektorokat, amelyek komponenseinek összege i, ahol $i = 0, 1, 2, \ldots$. Az adott i összegű vektorokból véges sok van, ezeket pl. lexikografikus sorrendben vehetjük.

Az előző fejtegetés szerint a H_f alakú halmazok, vagyis a diofantikus halmazok rekurzíve felsorolhatók. Olyan eljárást adtunk, ami a H_f elemeit elfogadja és

a H_f-en kívüli bemenetekre sosem áll meg. A Matijaszevics által betetőzött erőfeszítések fő eredménye a fordított irányú állítás:

Tétel (Matijaszevics tétele): *Ha egy $H \subseteq \mathbb{Z}^+$ halmaz rekurzíve felsorolható, akkor diofantikus is.*

□

A tétel bizonyításától itt el kell tekintenünk. Ha tehát $H \subseteq \mathbb{Z}^+$ rekurzíve felsorolható, akkor van olyan f polinom, melyre $H = H_f$. Az f polinom választható úgy, hogy $m = 14$ és $\deg f \leq 4$ is teljesüljön. A tétel szerint a polinomokra épülő számítási modell felismerőképessége megegyezik a Turing-gépekével.

Matijaszevics tételéből elég egyszerűen adódik a tizedik probléma eldönthetetlensége. A bizonyításhoz hasznos lesz a számelmélet következő klasszikus gyöngyszeme:

Tétel (J. L. Lagrange, 1751): *Minden nemnegatív egész szám előáll négy egész szám négyzetének összegeként.* □

Ezek után megmutatjuk, hogy ha volna algoritmus a diofantikus egyenletek megoldhatóságának véges sok lépésben való eldöntésére, akkor az L_h megállási nyelv is eldönthető lenne.

Tétel: *A diofantikus egyenletek megoldhatósága algoritmikusan eldönthetetlen.*

Bizonyítás: A megoldható egyenletekből (illetve azok leírásaiból) álló nyelv rekurzíve felsorolható. Ez ismét csak a számokból álló m-esek (ahol m az egyenlet változóinak a száma) felsorolásán alapuló próbálgatással adódik. Ha az egyenlet megoldható, akkor a szisztematikus behelyettesítgetés előbb-utóbb megoldást ad.

Azt kell ezután megmutatnunk, hogy nincs olyan algoritmus, amely mindig megáll, és éppen a megoldható egyenleteket fogadja el. Tegyük fel indirekte, hogy létezik egy ilyen M módszer.

Legyen $H \subset \mathbb{Z}^+$ az L_h nyelvnek megfelelő számhalmaz. H rekurzíve felsorolható, ezért Matijaszevics tétele szerint diofantikus is. Van olyan $f(x_1, \ldots, x_{14}, y)$ polinom, hogy $v \in \mathbb{Z}^+$ éppen akkor tartozik H-ba, ha az $f(x_1, \ldots, x_{14}, v) = 0$ egyenlet megoldható a *nemnegatív egészek* körében. (Ennek az egyenletnek a változói $x_1, x_2, \ldots, x_{14}$.) Lagrange tételét figyelembe véve ez egyenértékű azzal, hogy a következő 56 változós egyenlet megoldható-e az *egészek* között:

$$f(p_1^2 + q_1^2 + r_1^2 + s_1^2, p_2^2 + q_2^2 + r_2^2 + s_2^2, \ldots, p_{14}^2 + q_{14}^2 + r_{14}^2 + s_{14}^2, v) = 0.$$

Az M eljárás véges sok lépésben eldönti ennek az egyenletnek a megoldhatóságát, így a $(v \in H?)$ kérdést is; ez pedig ellentmondás, hiszen L_h nem rekurzív. □

Érintünk még egy érdekes problémakört, ami számhalmazoknak polinomok értékeiként való megadásával kapcsolatos. Az utóbbi századokban sokan próbáltak explicit formulákat adni bizonyos nevezetes sorozatokra. Így például meglehetősen népszerűek voltak a prímszámokat előállító képletek. Egy jellegzetes ilyen eredmény Leonhard Euler 1772-ből származó észrevétele, miszerint $n^2 - n + 41$ prímszám, ha $0 \leq n \leq 40$.[3]

Matijaszevics tételének van egy ilyesforma előállíthatósággal kapcsolatos szép következménye. Nevezzük a $H \subseteq \mathbb{Z}^+$ halmazt *explicitnek*, ha H előáll mint egy alkalmas egész együtthatós m-változós $g \in \mathbb{Z}[x_1, \ldots, x_m]$ polinomnak a nemnegatív egész helyeken felvett nemnegatív értékeinek H^g halmaza:

$$H = H^g := \left\{ u \in \mathbb{Z}^+ \;\middle|\; \begin{array}{l} \text{vannak olyan } u_1, \ldots, u_m \text{ természetes számok,} \\ \text{hogy } \; u = g(u_1, \ldots, u_m) \end{array} \right\}.$$

A definícióból az $(u_1, u_2, \ldots, u_m)$ vektorok felsorolásán alapuló érveléssel azonnal következik, hogy az explicit halmazok rekurzíve felsorolhatók. Matijaszevics tételéből egyszerűen adódik, hogy a fordított állítás is igaz.

Következmény: *A rekurzíve felsorolható halmazok explicitek.*

Bizonyítás: Legyen $H \subseteq \mathbb{Z}^+$ egy rekurzíve felsorolható halmaz. Matijaszevics tétele szerint alkalmas $f(x_1, x_2, \ldots, x_{14}, y)$ polinommal $H = H_f$ teljesül. Legyen ezután a $g \in \mathbb{Z}[x_1, \ldots, x_{14}, y]$ a következő polinom:

$$g(x_1, \ldots, x_{14}, y) = (y+1)(1 - f^2(x_1, \ldots, x_{14}, y)) - 1.$$

Ha $f(u_1, \ldots, u_{14}, v) \neq 0$, és $v \geq 0$, akkor $g(u_1, \ldots, u_{14}, v) < 0$, így nemnegatív értékeket $v \geq 0$ esetén g csak az f zérushelyeinél vehet fel. Ha pedig $f(u_1, \ldots, u_{14}, v) = 0$, akkor $g(u_1, \ldots, u_{14}, v) = (v+1)(1-0) - 1 = v$. A g nemnegatív értékei a természetes számokon tehát éppen a H_f elemei. □

A prímszámok halmaza rekurzív, hiszen a prímség eldönthető algoritmussal. Következésképpen a prímszámok halmaza explicit; ilyen értelemben tehát előállítható „képlet" segítségével. Ténylegesen meg is adtak ilyen polinomokat. A kisebbek ezek közül féltenyérnyi helyen leírhatók. Viselkedésük azonban túl bonyolult

[3] Euler polinomja csúcstartónak számít. Az első 41 természetes számnál mindenütt prím értéket vesz fel, mégpedig csupa különbözőt. Az 1 főegyütthatójú másodfokú polinomok között ez a leghosszabb ilyen sorozat. Ha más főegyüttható is megengedett, akkor valamivel hosszabb sorozatok is elérhetők. Például a $36x^2 - 810x + 2753$ (ún. Ruby-polinom) első 45 értéke prímszám.

ahhoz, hogy a gyakorlatban hasznavehetőek legyenek például prímek szisztematikus előállítására (csúnya, ámde népszerű szakszóval: generálására).

Az explicit halmazok értelmezésében két helyen is szerepel a nemnegativitás. Ezek közül az egyik nem lényeges.

Feladat: Mutassuk meg, hogy az explicit halmazok definíciójában az $u_i \in \mathbb{Z}^+$ feltétel helyettesíthető a gyengébb $u_i \in \mathbb{Z}$ feltétellel. (Lagrange tétele használható.)

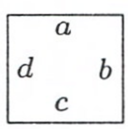
A definíció másik érdekes eleme, hogy nem a g teljes értékkészlete a H halmaz, csak a nemnegatív u értékek számítanak. Ez a feltétel már nem engedhető el:

Feladat: Tegyük fel, hogy a g polinom értékei között van két különböző prímszám, p és q. Ekkor van g-nek olyan értéke is, ami pq-val osztható, következésképpen g értékkészlete nem lehet azonos a prímek halmazával. (Tegyük fel, hogy $g(u_1, \ldots, u_m) = p$ és $g(v_1, \ldots, v_m) = q$, ahol $u_i, v_i \in \mathbb{Z}$. Legyen w_i olyan egész, melyre $w_i \equiv u_i \pmod{p}$ és $w_i \equiv v_i \pmod{q}$. Ekkor $g(w_1, \ldots, w_m)$ osztható pq-val.)

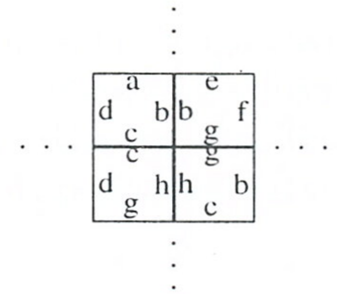
7.8.3. A Dominóprobléma

A kirakós játékok nehezek. Aki próbált már mondjuk egy 1500 darabkára gondosan felszabdalt képet összerakni, alighanem egyetért ezzel. Most egy olyan egyszerűen megfogalmazható, kirakó-típusú feladattal ismerkedünk meg, amelynél a megoldhatóság reménytelenül nehéz, pontosabban algoritmikusan eldönthetetlen problémát jelent.

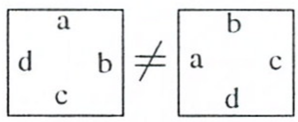
Dominón olyan egységnyi élű négyzet alakú lapot értünk, melynek a négy élére egy-egy jel van írva. Ezekről a jelekről a kényelem kedvéért feltesszük, hogy $\{0,1\}^*$-beli szavak.



A dominó típusát az e jelekből álló rendezett négyes jelenti. Az előző rajz dominójának a típusa (a, b, c, d). A dominókat a jól ismert szabály szerint lehet egymás mellé rakni: az illeszkedő élek mentén azonos jeleknek kell szerepelni.



A dominókat nem szabad forgatni. Az (a, b, c, d) típusú dominót csak úgy tehetjük le, hogy a felső él mentén a legyen.

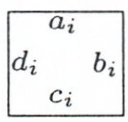


A Dominóprobléma ezek után a következő: *Adott dominó-típusok egy véges $\mathcal{F}$ halmaza; eldöntendő, hogy a sík lefedhető-e hézagtalanul szabályosan illeszkedő $\mathcal{F}$-beli típusú dominókkal.* Természetesen a fedéshez az egyes típusokból végtelen sok példány használható.

A k típusból álló $\mathcal{F}$ készlet leírható a következő tagozódású F szóval:

$$F := x_1\# \dots \#x_k\#,$$

ahol az $x_i = a_i * b_i * c_i * d_i$ szó ($a_i, b_i, c_i, d_i \in \{0,1\}^*$) az

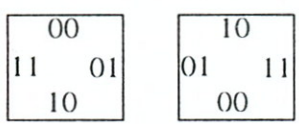


dominó típusának a leírása.

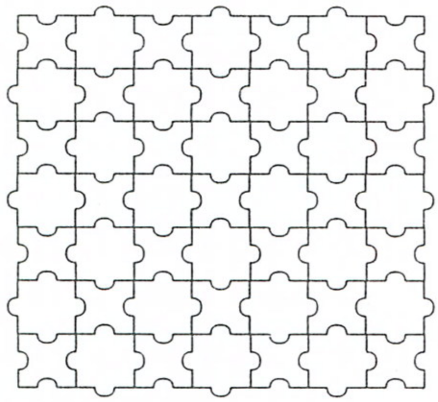
A Dominófeladatnak megfelelő D nyelv az alábbi:

$$D = \{F \in \{0, 1, *, \#\}^*;\ \text{a sík lefedhető } \mathcal{F}\text{-típusú dominókkal}\ \}.$$

Példa: Az $F = 00 * 01 * 10 * 11\#10 * 11 * 00 * 01\#$ szó a következő két típusból álló készlet kódja.



Ezzel a készlettel a sík kirakható, másként mondva $F \in D$. A kétféle dominóval a sakktábla fekete és fehér mezőinek váltakozását követő alakzatban parkettázhatjuk ki a síkot. A megoldáson túl az illeszkedési szabályból adódó kötöttségeket is szemlélteti az alábbi – XV. századi portugál padlómozaikról ellesett – minta:



A Dominóprobléma nem oldható meg algoritmussal. Ezt mondja ki a következő tétel. A terjedelmes, de nem különösebben nehéz bizonyítást mellőzzük.

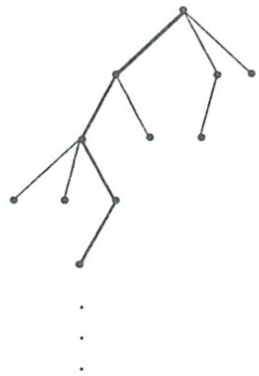
Tétel: *A D nyelv nem rekurzív.* □

Az előző tétel szerint nem lehetséges, hogy D és a komplementere $\{0, 1, *, \#\}^* \setminus D$ is rekurzíve felsorolható legyen. Az utóbbi nyelvről viszont megmutatjuk, hogy rekurzíve felsorolható.

Tétel: *$D \in co\mathcal{RE}$, azaz $\{0, 1, *, \#\}^* \setminus D \in \mathcal{RE}$.*

A tétel bizonyításában hasznunkra lesz egy nevezetes, ám egyszerű tény. Az állítás azzal a kijelentéssel fogalmazható meg szemléletesen, hogy *ha az emberiség sohasem hal ki, akkor létezik olyan dinasztia, amely sohasem szakad meg.* Pontosabban ez így mondható:

Lemma (König-lemma): *Legyen G egy végtelen, gyökeres irányított gráf, amelynek csúcshalmaza a természetes számokkal sorszámozott részekre (szintekre) osztható úgy, hogy:*
(0) A nulladik szinten egyedül a gyökér van. Egy csúcsból csak az eggyel nagyobb sorszámú szintre mehet él.
(1) Minden egyes szinten csak véges sok csúcs van.
(2) Minden – a gyökértől különböző – csúcsnak van őse abban az értelemben, hogy fut bele él az eggyel alacsonyabb sorszámú szintről.
Ekkor a gráfban van végtelen irányított út.



Bizonyítás: Nevezzük a gráf y csúcsát az x csúcs *leszármazottjának*, ha y az x-ből elérhető irányított úton. Jelölje x_0 a gráf gyökerét. A (2) és (0) feltételek miatt a G minden csúcsába vezet út x_0-ból. Mivel a gráf végtelen, ebből arra jutunk, hogy az x_0 gyökérnek végtelen sok leszármazottja van. Ezután indukcióval megadunk egy $y_0, y_1, \ldots, y_i, \ldots$ végtelen G-beli utat. Legyen $y_0 = x_0$. Tegyük fel ezután, hogy az $y_0, y_1, \ldots, y_i$ csúcsokat már kiválasztottuk úgy, hogy y_j a j-edik szintről való, és végtelen sok leszármazottja van $(0 \leq j \leq i)$, továbbá (y_j, y_{j+1}) éle a gráfnak, ha $(0 \leq j < i)$. Ezek a feltételek adják az indukciós feltevést. Ez nyilván teljesül $i = 0$-ra.

Azt kell csak megmutatnunk, hogy a feltétel az y_{i+1} alkalmas választásával örökíthető i-ről $i+1$-re. Ez pedig egyszerű: legyen y_{i+1} az y_i egy olyan gyermeke (az y_i-vel összekötött csúcs az $i+1$. szintről), melynek végtelen sok leszármazottja van. Az indukciós feltevés szerint y_i-nek végtelen sok leszármazottja van. Az (1) kikötés miatt viszont csak véges sok gyermeke lehet; van tehát olyan gyermeke is, amelynek végtelen sok leszármazottja van.

Ilyen választással az $y_0, y_1, \ldots, y_{i+1}$ sorozatra is igaz lesz az indukciós feltevés. A sorozat definíciójával tehát sehol sem akadunk el; $y_0, y_1, \ldots, y_i, \ldots$ tényleg egy végtelen út. □

A tétel bizonyítása: Álljon $L \subseteq \{0, 1, *, \#\}^*$ nyelv azokból a szavakból, amelyek kódjai valamilyen dominókészletnek. Világos, hogy L rekurzív: a helyes felépítésű leírások mindig megálló algoritmussal felismerhetők. Elég ezek után azt igazolni, hogy az $L \setminus D$ nyelv rekurzíve felsorolható.

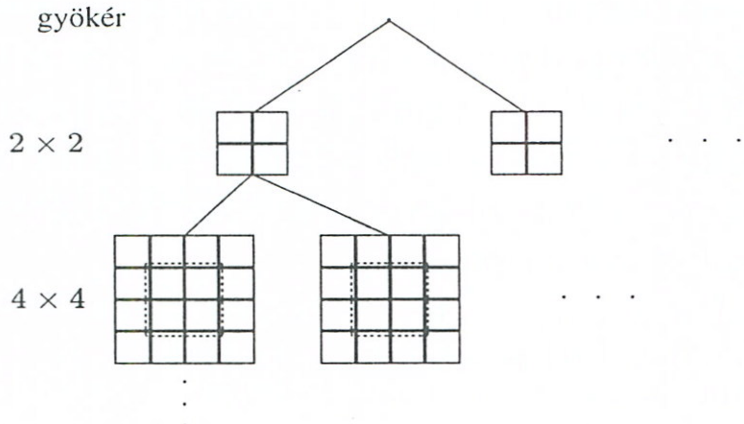
Álljon D^* azon $\mathcal{F}$ dominókészletek leírásaiból, melyekhez van olyan n, hogy már a $2n \times 2n$-es négyzet sem rakható ki $\mathcal{F}$-beli dominókkal. Egyszerű, és mostanra már bizonyára ismerős érvelés mutatja, hogy $D^* \in \mathcal{RE}$. Próbáljuk ugyanis sorban $n = 1, 2, \ldots$-re $\mathcal{F}$-beli dominókkal kirakni a $2n \times 2n$-es négyzetet. Ezt adott n-re – ha máshogy nem – az összes kitöltési lehetőség végignézésével tehetjük meg. Ha van olyan n, melyre a $2n \times 2n$-es négyzet nem parkettázható ki, akkor ezen az úton ez véges sok, bár esetleg csillagászati számú lépés után kiderül. Ekkor megállunk elfogadva az $\mathcal{F}$ készletet leíró F szót.

Ezután a tétel igazolásához elegendő belátni, hogy

$$D^* = L \setminus D.$$

Világos, hogy $D^* \subseteq L \setminus D$, hiszen D^* dominókészletek leírásaiból áll, továbbá D^*-beli készletekkel a sík nem rakható ki, hiszen már egy alkalmas négyzet sem rakható ki.

A fordított irányú $D^* \supseteq L \setminus D$ tartalmazás belátásához elég megmutatni, hogy ha egy $\mathcal{F}$ készlet nincs D^*-ban, akkor nincs $L \setminus D$-ben sem. Egyszerűbben fogalmazva: ha minden n-re a $2n \times 2n$-es négyzet lefedhető az $\mathcal{F}$ készlet dominóival, akkor az egész sík is. Ennek igazolására egy gyökeres, végtelen, szintezett gráfot definiálunk. A gráf csúcsai a négyzetek szabályos kitöltéseinek felelnek meg a következők szerint. A nulladik szinten van a gráf gyökere; ez felel meg az üres négyzet (üres) fedésének. Általában az n-edik szint csúcsai a $2n \times 2n$-es négyzet szabályos kitöltései $\mathcal{F}$-beli dominókkal. Ezzel megadtuk a gráf csúcsait. A $2n \times 2n$-es négyzet egyik kitöltésétől akkor vezet él a $2(n+1) \times 2(n+1)$-es négyzet valamely kitöltéséhez, ha utóbbinak a középső $2n \times 2n$-es négyzetét fedő része megegyezik az előbbivel.



Egyszerű ellenőrizni, hogy a gráf teljesíti a König-lemma feltételeit. A gráf azért lesz végtelen, mert tetszőlegesen nagy négyzetek kirakhatók $\mathcal{F}$-beli dominókkal. A lemma szerint van ebben a gráfban végtelen irányított út. Ez az út pedig az egész sík egy kirakását adja. □

7.8.4. Post megfeleltetési problémája

A következő – *Emil Posttól* származó – eldönthetetlen probléma is egyfajta kirakós feladat, de nem geometriai, hanem nyelvészeti jellegű. Azért szólunk róla, mert felettébb hajlékony eszközt nyújt a matematikai és számítógépes nyelvészet területén egy sereg probléma eldönthetetlenségének a bizonyítására.

Legyen Σ egy véges *abc*. *Post megfeleltetési problémájának* egy példánya (bemenete) egy (s, t) $(s, t \in \Sigma^*)$ alakú rendezett párokból álló véges $\mathcal{P}$ halmaz. A megfeleltetési feladat $\mathcal{P}$ bemenetét *megoldhatónak* nevezzük, ha vannak olyan (nem feltétlenül különböző) $\mathcal{P}$-beli $(s_1, t_1), (s_2, t_2), \ldots, (s_n, t_n)$ párok úgy, hogy

$$s_1 s_2 \cdots s_n = t_1 t_2 \cdots t_n.$$

A feltétel tehát az, hogy a kiválasztott párok első komponenseinek összeolvasásával adódó szó egyezzen meg a második komponensek egymás után fűzésével kapott szóval. Ilyenkor az $s_1 s_2 \cdots s_n$, vagy ami ugyanaz, a $t_1 t_2 \cdots t_n$ szót a $\mathcal{P}$ *megoldásának* nevezzük. Például a $\mathcal{P} = \{(iz, riz), (kar, ka), (ma, ma)\}$ rendszer megoldható. Egy lehetséges megoldás a *karizma* szó.

Feladat: Mutassuk meg, hogy a $\mathcal{P} = \{(ab, a), (ab, ba), (b, ba)\}$ rendszernek nincs megoldása.

A következő feladatból kitűnik, hogy a megoldáshoz tényleg szükség lehet arra, hogy bizonyos párokat többször is felhasználjunk. Legyen $\mathcal{P}' = \{(aa, aab), (bb, ba), (abb, b)\}$. A $\mathcal{P}'$ bemenet tehát három párból áll. Ugyanakkor igaz a következő.

Feladat: Mutassuk meg, hogy a $\mathcal{P}'$ rendszer megoldható, és a legrövidebb megoldás négy párból illeszthető össze (azaz 4 a legkisebb n, amivel megoldást kaphatunk).

A probléma angol nevéből eredően (Post's correspondence problem) PCP jelöli a megfeleltetési feladat megoldható példányainak leírásaiból álló nyelvet. Megmutatható, hogy nincs algoritmus, ami a megfeleltetési feladat megoldható példányait felismerné. *A PCP nyelv tehát nem rekurzív.* Érvényes másfelől a következő:

Feladat: Mutassuk meg, hogy a PCP nyelv rekurzíve felsorolható. (Tegyük fel, hogy a megfeleltetési feladat $\mathcal{P}$ bemenete k párból áll. Vegyük sorra $(n = 1, 2, 3, \ldots)$ az n hosszú sorozatokat, melyek tagjai az $1, 2, \ldots, k$ sorszámok, és ellenőrizzük, hogy a sorozatnak megfelelő párok adnak-e megoldást.)

7.8.5. Egy nyitott kérdés: a kongruens számok felismerése

Az eldönthetetlenség témakörétől búcsúzva meg szeretnénk jegyezni, hogy nem egy lezárt kutatási területről van szó. Ennek érzékeltetésére egy olyan problémát ismertetünk, amelyre eddig nem találtak algoritmust, és az eldönthetetlenségét sem sikerült bizonyítani. A probléma ártatlanul egyszerűnek tűnik, és bizonyos szempontból igen régi. A fogalom, amihez kapcsolódik, már a X. század arab írástudóinak a munkáiban is megtalálható.

Definíció: *Az m pozitív egész* kongruens szám, *ha van olyan derékszögű háromszög, melynek a területe m, és az oldalai racionális hosszúságúak.*

Megmutatható, hogy 1, 2, 3 és 4 nem kongruens számok. Már e tények igazolása sem túl egyszerű. Kis könnyebbség, hogy a négyesre vonatkozó állítás azonnal következik az egyesről szólóból (és fordítva). Például, ha volna a négyeshez megfelelő derékszögű háromszög, akkor azt a felére kicsinyítve egységnyi területű háromszöget kapnánk.

Feladat: Használva, hogy az $x^4 + y^4 = z^2$ diofantikus egyenletnek nincs csupa pozitív megoldása[4], mutassuk meg, hogy a 2 nem kongruens szám. (Ha 2 kongruens szám volna, akkor léteznének olyan u, v, w pozitív egészek, melyekre $u^2 + v^2 = w^2$, továbbá u és v relatív prímek, és uv négyzetszám. Az utóbbi két feltétel miatt u és v is négyzetszámok.)

Mint azt Leonardo Pisano – ismertebb nevén: Fibonacci – 1220 körül megállapította, az 5 kongruens szám: $3/2$, $20/3$ és $41/6$ a legegyszerűbb olyan racionális derékszögű háromszög oldalai, amelynek a területe 5. A sokszor látott $3, 4, 5$ hármas mutatja, hogy 6 is kongruens szám.

Felmerül a kérdés, van-e általános recept a kongruens számok felismerésére. Ilyet eddig nem sikerült találni. Nem ismert, hogy a kongruens számokból álló nyelv rekurzív-e[5].

Feladat: Mutassuk meg, hogy a kongruens számokból álló nyelv rekurzíve felsorolható. (Legyen n a bemenet. Generáljuk valamilyen sorrendben a pozitív racionális számokból álló (p, r, s) hármasokat. Egy hármasról ellenőrizzük, hogy $p^2 + q^2 = r^2$ és $pq/2 = n$ teljesülnek-e. Ha igen, akkor elfogadjuk n-et. Ellenkező esetben folytatjuk a következő hármassal.)

[4] Az $x^4 + y^4 = z^2$ egyenlet ilyen értelmű megoldhatatlanságát Pierre Fermat igazolta.

[5] Az aritmetikai geometria egy nevezetes sejtéséből, a Birch–Swinnerton-Dyer-sejtésből következik, hogy a kongruens számok problémája eldönthető. Ez a hipotézis bizonyos harmadfokú síkgörbék racionális koordinátájú pontjairól szól. Jerrold Tunnell bizonyította, hogy ha a sejtés igaz, akkor a kongruens számoknak van hatékony jellemzése. Ekkor a páratlan négyzetmentes n egész pontosan akkor kongruens szám, ha a $2x^2 + y^2 + 8z^2 = n$ diofantikus egyenletnek kétszer annyi megoldása van, mint a $2x^2 + y^2 + 32z^2 = n$ egyenletnek. Ebből rögvest látszik, hogy legfeljebb $(n+1)^3$ számú (x, y, z) egész vektor vizsgálatával eldönthető, hogy n kongruens-e. Tunnell hasonló jellemzést adott a páros n-ekre is.

7.9. Kolmogorov-bonyolultság

Rosencrantz: *Fej.* (Megismétlik.)
Fej. (Zavart nevetéssel Guildensternre tekint.)
Kezd egy kicsit unalmas lenni, nem?
TOM STOPPARD: Rosencrantz és Guildenstern halott

Itt az információtartalom egy algoritmusokon alapuló megközelítésével foglalkozunk, amit Gregory J. Chaitin, Andrej N. Kolmogorov és Ray J. Solomonoff dolgoztak ki a hatvanas években. Kiindulásul próbáljuk megfogalmazni, hogy egy I^*-beli szó mennyire tömören írható le; másképpen fogalmazva: milyen rövid, milyen kevés jelből álló kóddal nyomható össze optimálisan. Az ilyen fogalomalkotó kísérletek egy lehetséges buktatójára világít rá az *írógép-paradoxon*:

Tekintsük azokat a természetes számokat, amelyeket magyar nyelven legfeljebb 100 billentyűleütéssel definiálni lehet. A billentyűk száma véges, így ezen számok halmaza is véges. Van tehát egy legkisebb természetes szám, amit nem lehet definiálni a fenti módon. De ekkor „az a legkisebb szám, amely nem definiálható magyar nyelven legfeljebb száz billentyűleütéssel" egy száznál rövidebb jelsorozat, tehát olyan számot határoz meg, amely mégis benne van a definiálható számok halmazában, ami nyilvánvaló képtelenség.

A paradoxon arra figyelmeztet bennünket, hogy a „röviden való leírhatóság" csak úgy önmagában még nem lesz használható fogalom. Egy lehetséges kritika, hogy a *magyar nyelven leírhatóság* nem köti meg eléggé a leírás értelmezésének módját. Kössük ki tehát, hogy a leírás értelmezését, más szóval a dekódolást algoritmussal, pontosabban – a Church–Turing-tézis alapján – Turing-géppel végezzük.

Milyen Turing-gépet használjunk erre a célra? Természetes elvárás, hogy a matematikai képlettel tömören leírható számok hatékonyan kódolhatók, jelentősen összenyomhatók legyenek. Például az $n = 2^k - 10$ alakú számok bináris kódja $k = \log_2 n$ hosszúságú, ha $k > 4$. Viszont a $2^k - 10$ kifejezés hossza csak a $2^x - 10$ képlet hosszából valamint k bináris hosszából tevődik össze, ami $\log_2 \log_2 n +$ konstans. Szeretnénk, ha a fogalom tükrözné az ilyen alakú n számok rövid leírhatóságát. Sőt, olyan jellegű számokat is hatékonyan szeretnénk kódolni, mint a k-adik Fibonacci-szám, a k-adik prímszám, a π első k számjegye, stb. A leírást kibontó gépnek ezek szerint „értenie" kell az egyszerű aritmetikai képleteknél általánosabban az algoritmusokat is. Ennek a feltételnek eleget tesznek az univerzális Turing-gépek.

Rögzítsünk tehát egy U univerzális Turing-gépet, és értelmezzük az $x \in I^*$ szó bonyolultságát mint a legrövidebb $y\#z$ input szó hosszát, melyre U az x szót

számítja ki. Az így adódó fogalom meglepően értelmesnek bizonyul. Az U gép választásától nagy mértékben független, és aszimptotikus értelemben jó (konstans eltérésen belüli) közelítését adja az „optimumnak". Például az $n = 2^k - 10$ alakú számok bonyolultsága, mint várjuk, a $2^x - 10$ képlet hosszától eltekintve általában $\log_2 \log_2 n$ lesz. Rövidebb akkor lehet, ha k speciális alakú (k-nak $\log_2 k$-nál rövidebb leírása van). Azt is igazolni tudjuk, hogy a legtöbb esetben k leírása nem lehet $\log_2 k$-nál sokkal rövidebb.

Legyen továbbra is $I = \{0, 1\}$. Olyan Turing-gépekre szorítkozunk, amelyeknek a bemenő *abc*-je I, és a számításaik eredménye is I-beli betűkből áll. Egy ilyen M gép a korábbiakkal összhangban az $f_M : I^* \to I^*$ parciális függvényt számolja ki. Jelöljük $C_M(x)$-szel annak a legrövidebb bemenő szónak a hosszát, mellyel elindítva M az x szót adja eredményül:

$$C_M(x) = \begin{cases} \min\{|y| : y \in I^*,\ f_M(y) = x\} & \text{ha ilyen } y \text{ létezik,} \\ \infty & \text{különben.} \end{cases}$$

A $C_M(x)$ szám méri, hogy x mennyire nyomható össze akkor, ha a kibontást, vagyis az összenyomott szó visszafejtését az M algoritmus végzi: $C_M(x)$ a legtömörebb kódszó hossza, mely az M algoritmus számára x-et kódolja. A $C_M(x)$ érték erősen függ M-től. Az x ismeretében könnyen szerkeszthetünk olyan M_1 gépet, mely az üres bemeneten x-et számolja ki. Olyan M_2 gép is van, amely sosem adja eredményül x-et. Ekkor $C_{M_1}(x) = 0$ és $C_{M_2}(x) = \infty$.

Most megmutatjuk, hogy az univerzális Turing-gépek körében a C érték nem függ ennyire vadul a gép választásától. Ki fog derülni, hogy a kibontó képességet illetően egy univerzális gép semelyik másik gépnél sem lehet számottevően rosszabb. Emlékeztetjük az olvasót, hogy egy U univerzális gép bemenete $w\#s$ alakú, ahol $w, s \in I^*$. Az érdekes esetekben w egy M Turing-gép leírása, amit M_w-vel is jelölünk. Az U gép a $w\#s$ bemeneten utánozza az M_w működését az s inputtal. A továbbiakban feltesszük, hogy az univerzális gépeink is egyszalagosak, és a szalag-*abc*-jük $\{0, 1, \ddot{u}\}$. Ennek megfelelően az $\#$ elválasztó jelet egy véges bitsorozat kódolja.

Tétel (invariancia-tétel): *Legyen U egy univerzális Turing-gép. Ekkor tetszőleges M Turing-gépre létezik egy (csak M-től függő) $c_M \in \mathbb{Z}^+$ állandó, mellyel minden $x \in I^*$ szóra teljesül a következő egyenlőtlenség:*

$$C_U(x) \le C_M(x) + c_M.$$

Bizonyítás: Tegyük fel, hogy az M gép leírása a $w \in I^*$ szó, és legyen y egy legrövidebb szó, amiből M az x-et bontja ki: $y \in I^*$, $f_M(y) = x$, és $|y| = C_M(x)$. Az univerzális gép definíciója szerint ekkor U a $w\#y$ bemeneten x-et

adja eredményül; más szóval $f_U(w\#y) = x$. Ebből a C_U függvény értelmezése alapján következik, hogy

$$C_U(x) \leq |w\#y| = |w\#| + |y| = |w\#| + C_M(x).$$

A $c_M = |w\#|$ választás tehát megfelel a követelményeknek. □

Az invariancia-tétel szerint a $C_U(x)$ érték csak legfeljebb egy x-től független állandóval haladhatja meg $C_M(x)$-et. Ezt úgy értelmezhetjük, hogy az U-hoz tartozó összenyomhatóság mértéke nem marad el számottevően semelyik másik M gép szerinti összenyomhatóságtól sem.

Következmény: *Legyenek U_1 és U_2 univerzális Turing-gépek, melyek input abc-je $I = \{0, 1\}$. Ekkor van olyan $c = c_{U_1,U_2}$ állandó, hogy minden $x \in I^*$ szóra*

$$|C_{U_1}(x) - C_{U_2}(x)| \leq c.$$

Bizonyítás: Alkalmazzuk az invariancia-tételt először az $U = U_1$, $M = U_2$, majd pedig az $U = U_2$, $M = U_1$ szereposztással. Adódik, hogy tetszőleges $x \in I^*$ szóra teljesülnek a $C_{U_1}(x) \leq C_{U_2}(x) + c_{U_2}$ és $C_{U_2}(x) \leq C_{U_1}(x) + c_{U_1}$ egyenlőtlenségek, ahol c_{U_i} az U_i-től függő pozitív állandó. Arra jutunk ezekből, hogy a $c = \max\{c_{U_1}, c_{U_2}\}$ értékkel $|C_{U_1}(x) - C_{U_2}(x)| \leq c$ teljesül. □

Az előző szerint $C_U(x)$ nem függ nagyon erősen az U univerzális gép választásától. Ha U' egy másik univerzális gép, akkor a $C_U(x)$ és $C_{U'}(x)$ eltérése egy x-től független korláton belül marad. Ez alapot ad arra, hogy egyetlen – mostantól fogva rögzített – U univerzális gép segítségével tanulmányozzuk az összenyomhatóságot.

Definíció (Kolmogorov-bonyolultság):
Legyen $x \in I^*$. A $C(x) := C_U(x)$ mennyiség az x szó Kolmogorov-bonyolultsága.

Egy függvény definícióját látva az embernek olykor kedve támad, hogy kiszámolja néhány helyettesítési értékét. Mi lesz például $C(0010)$? Kíváncsiságunk mindjárt akadályba ütközik: az U-t nem adtuk meg elég pontosan ahhoz, hogy a meghatározást követve számolhassunk vele. Rövidesen kiderül, hogy $C(x)$ kiszámítása nehéz feladat; a C függvény nem rekurzív. Mihez lehet hát kezdeni egy ilyen függvénnyel, amelynek az értékei és számolhatósága körül ennyi gond van? A válasz az, hogy $C(x_n)$ alakú sorozatok növekedési rendjét érdemes vizsgálni, ahol $x_1, x_2, \ldots$ növekvő hosszúságú I^*-beli szavak sorozata. Például az $n = 2^k - 10$ alakú számokra $C(n) \leq \log_2 \log_2 n + c'$ teljesül alkalmas c' állandóval. Az egyenlőtlenség bal oldalán $n \in I^*$ az n szám szokásos bináris kódját jelöli. Az egyenlőtlenség az invariancia-tétel egyszerű következménye. Van ugyanis

olyan algoritmus, amely a binárisan írt, és ezért legfeljebb $\log_2 \log_2 n + 1$ hosszúságú k-ból n-et számítja ki. A megfelelő M géppel tehát $C_M(n) \leq \log_2 \log_2 n+1$, és innen az invariancia-tétel adja a $C(n)$-re vonatkozó állítást.

Az invariancia-tétel további egyszerű következménye, hogy egy I^*-beli szó Kolmogorov-bonyolultsága nem haladja meg lényegesen a szó hosszát:

Következmény: *Legyen $x \in I^*$. Ekkor $C(x) \leq |x|+k$, ahol k egy x-től független állandó.*

Bizonyítás: Jelölje M azt a Turing-gépet, amely az inputot érintetlenül hagyva egy lépésben megáll. Nyilván $f_M(x) = x$, sőt $C_M(x) = |x|$ is teljesül bármely $x \in I^*$ szóra. Az M gépre alkalmazhatjuk az invariancia-tételt; éppen a kívánt egyenlőtlenséget kapjuk. □

A fordított irányt illetően megmutatjuk, hogy a szavak túlnyomó többségének a Kolmogorov-bonyolultsága közel van a szó hosszához. Vannak olyan szavak is, amelyek nem írhatók le rövidebben, mint a saját hosszuk. Ezeket külön névvel illetjük:

Definíció: *Az $x \in I^*$ szó* összenyomhatatlan, *ha $C(x) \geq |x|$.*

Tétel: *Legyen $k \in \mathbb{Z}^+$. Legfeljebb $2^{k+1} - 1$ $x \in I^*$ szó van, melyre $C(x) \leq k$. Következésképpen minden $n \geq 1$ egészre létezik n hosszúságú összenyomhatatlan szó. Ha $n > 8$, akkor az n hosszú I^*-beli szavak több, mint 99 százalékának a Kolmogorov-bonyolultsága nagyobb, mint $n - 8$.*

Bizonyítás: Legyen

$$H_k = \{x \in I^* : \ C(x) \leq k\}.$$

Ha $x, x' \in H_k$, akkor a Kolmogorov-bonyolultság definíciója szerint vannak olyan $y, y' \in I^*$ szavak, melyekre $f_U(y) = x$, $f_U(y') = x'$, továbbá $|y| \leq k$ és $|y'| \leq k$. Nyilvánvaló, hogy ha $x \neq x'$, akkor $y \neq y'$, hiszen U egy bemenetéhez nem tartozhat két különböző eredmény. A H_k halmaznak eszerint legfeljebb annyi eleme lehet, mint ahány k-nál nem hosszabb szó van I^*-ban. Az utóbbi szavak száma pedig $1 + 2 + \cdots + 2^{k-1} + 2^k = 2^{k+1} - 1$. Ezzel az első állítást igazoltuk.

A második a $k = n - 1$ választással azonnal adódik az elsőből: az n hosszú szavak száma 2^n, a H_{n-1} halmazban pedig legfeljebb $2^n - 1$ szó van. Létezik tehát olyan n hosszú szó, ami nincs H_{n-1}-ben; ez egy összenyomhatatlan szó.

Hátra van még az utolsó állítás. A H_{n-8} halmaznak legfeljebb $2^{n-7} - 1 < 2^{n-7}$ eleme van. A kedvezőtlen esetek aránya az n hosszú szavak között így legfeljebb $2^{n-7}/2^n = 1/128$, ami kisebb, mint $1/100$. □

Feladat: Legyen $\omega : \mathbb{Z}^+ \to \mathbb{Z}^+$ egy végtelenhez tartó függvény, azaz $\lim_{n\to\infty} \omega(n) = \infty$ teljesüljön. Igazoljuk, hogy ekkor

$$\lim_{n\to\infty} \frac{|\{x \in I^* : |x| = n,\ C(x) > n - \omega(n)\}|}{|\{x \in I^* : |x| = n\}|} = 1.$$

Vagyis „majdnem minden" x szóra igaz a $C(x) > |x| - \omega(|x|)$ egyenlőtlenség. (Mutassuk meg, hogy a $|H_{n-\omega(n)}|/2^n$ hányados nullához tart.)

Egy tipikus szó Kolmogorov-bonyolultsága tehát közel van a hosszához. Ennek furcsa ellenpontjaként említhetjük, hogy mégsem tudunk minden n-re egy olyan n hosszú x_n szót *algoritmussal előállítani*, amelyre mondjuk $C(x_n) \geq n/2$. Ha volna ugyanis egy M Turing-gép, ami a binárisan írt n inputra egy ilyen x_n szót adna, akkor $C_M(x_n) \leq \log_2 n + 1$, és az invariancia-tétel alapján $C(x_n) \leq \log_2 n + c$ teljesülne. Az utóbbi mennyiség viszont elég nagy n-re kisebb lesz, mint $n/2$. Ez pedig ellentmond a $C(x_n) \geq n/2$ feltételnek. Ezek után már az sem meglepő, hogy a Kolmogorov-bonyolultságot nem lehet kiszámítani:

Tétel: *A $C : I^* \to I^*$ függvény nem rekurzív.*

Bizonyítás: A szokásos bináris ábrázolás segítségével a $\mathbb{Z}^+$ halmazt I^* részének tekinthetjük. Legyen ezután $F : \mathbb{Z}^+ \to I^*$ a következő függvény:

$$F(m) := \min\{x \in I^*; C(x) \geq m\},$$

ahol a minimumot az I^*-beli szavak kanonikus rendezése szerint értjük. Az F minden természetes számra értelmezett, mert létezik tetszőlegesen nagy Kolmogorov-bonyolultságú szó. Tegyük fel indirekte, hogy C rekurzív. Ezt a feltevést használva megmutatjuk, hogy F parciálisan rekurzív, azaz van olyan M Turing-gép, melyre $F = f_M$. Az M algoritmus munkája az m bemenetnél felettébb egyszerű: vesszük az $x \in I^*$ szavakat a kanonikus sorrendben, és kiszámítjuk a $C(x)$ értéket. Ezen a ponton aknázzuk ki a C rekurzivitását. Kiírjuk az első olyan x szót, amelyre $C(x) \geq m$, és megállunk. Mármost az F definíciója szerint

$$m \leq C(F(m)).$$

Másrészt az invariancia-tétel alapján alkalmas c-vel igaz, hogy

$$C(F(m)) \leq C_M(F(m)) + c.$$

Ugyanakkor a C_M értelmezése szerint, ha $m > 0$, akkor

$$C_M(F(m)) \leq \log_2 m + 1,$$

hiszen M a binárisan ábrázolt m inputból előállítja $F(m)$-et. A kapott egyenlőtlenségeket összerakva $m > 0$ esetén arra jutunk, hogy

$$m \leq \log_2 m + 1 + c,$$

ami elég nagy m-re lehetetlen. Az így nyert ellentmondás bizonyítja, hogy C nem rekurzív. □

A Kolmogorov-bonyolultság néhány alkalmazása

A Kolmogorov-bonyolultság önmagában is érdekes fogalom, de ezenfelül kitűnik még a sokoldalú, színes alkalmazási lehetőségeivel. Eme alkalmazások közül itt hármat említünk.

1. A Megállási probléma eldönthetetlensége

A Kolmogorov-bonyolultság tulajdonságait használva egyszerűen igazolható, hogy az L_h megállási nyelv eldönthetetlen. Tegyük fel ugyanis indirekte, hogy létezik egy az L_h nyelvet felismerő, mindig megálló M algoritmus. Megmutatjuk, hogy az M segítségével a C függvény kiszámítható, ami ellentmond az előző tételnek.

Legyen $x \in I^*$. A $C(x)$ érték az (egyik) legrövidebb olyan $y \in I^*$ szó hossza, melyre $f_U(y) = x$. Itt nyilván elegendő az olyan y szavakra szorítkozni, melyekkel mint bemenetekkel az U valamikor megáll. A $C(x)$ meghatározásához tehát a kanonikus sorrendben vesszük az $y \in I^*$ szavakat. Egy y szóra először az M algoritmussal ellenőrizzük, hogy U megáll-e az y inputtal. Ha a válasz nemleges, akkor eldobjuk y-t, és vesszük a következő szót. Ha a válasz igenlő, akkor az y-nal elindítjuk az U-t. Az U munkájának végeztével ellenőrizzük, hogy az $f_U(y)$ eredmény egyenlő-e x-szel. Ha nem, akkor a kanonikus sorrend szerint következő y-nal próbálkozunk. Az első olyan y szónál állunk meg, amelyre $f_U(y) = x$, és kiírjuk az y hosszát. Ilyen y létezik, és a vázolt recepttel véges sok lépésben meg is találjuk. □

Feladat: Mutassuk meg, hogy a

$$\{x\#n : \ x \in I^*, \ n \in \mathbb{Z}^+, \ C(x) \leq n\}$$

nyelv rekurzíve felsorolható, de nem rekurzív. (Az első állításhoz a $(\mathbb{Z}^+)^2$-beli párok algoritmikus felsorolása használható. A másodikhoz mutassuk meg, hogy ha a nyelv rekurzív lenne, akkor a C függvényt ki tudnánk számítani.)

2. Palindrómák felismerése egyszalagos Turing-géppel

Korábban láttuk, hogy egy k szalagos M Turing-gép szimulálható olyan egyszalagos N géppel, melynek az időfüggvénye legfeljebb az M időfüggvényének négyzetével arányosan nő: $T_N(n) = O(T_M^2(n))$. Most megmutatjuk, hogy ez a kvadratikus becslés éles: egy olyan nyelvet ismertetünk, mely kétszalagos géppel lineáris (azaz $O(n)$) időben felismerhető, egyszalagos géppel viszont nem ismerhető fel $o(n^2)$ lépésen belül.

Egy szót *palindrómának* vagy *tükörszónak* nevezünk, ha visszafelé olvasva is magát a szót kapjuk. Például a *kajak* egy tükörszó. Az egyik legnevezetesebb palindróma a Napóleon sírkövén levő felirat: ABLE WAS I ERE I SAW ELBA.

Legyen L_p az I^*-beli palindrómákból álló nyelv. Könnyen látható, hogy kétszalagos Turing-géppel egy n hosszú $w \in I^*$ szónak az L_p-be tartozása $O(n)$ lépésben eldönthető: először felmásoljuk a w input szót a második szalagra, majd az első fejjel visszamegyünk a szalag elejére. Végül a két fejjel ellentétes irányban haladva ellenőrizzük, hogy a tükrösen elhelyezkedő jelek egyeznek-e. Egyszalagos géppel már nem megy ilyen gyorsan:

Tétel: *Nincs olyan egyszalagos M Turing-gép, amely az L_p nyelvet ismeri fel, és amelynek időfüggvényére $T_M(n) = o(n^2)$ teljesül.*

Először megpróbálunk szemléletes képet adni a bizonyításban szereplő gondolatokról. Legyen M egyszalagos Turing-gép, amelynek a nyelve L_p és nézzük M munkáját egy n hosszú $s \in I^*$ input szón. Osszuk három egyenlő részre a szalagnak a bemenetet tartalmazó részét. Az ($s \in L_p$?) megválaszolásához a gépnek valahogy össze kell vetnie az s első harmadát az utolsóval. Az M konstans mennyiségű – mondjuk b bit – információt tud a belső állapotaiban tárolni, és ezért a fej mozdításával egy hellyel arrébb vinni. Az M munkáját úgy képzeljük el, hogy az s első harmadából való információdarabokat vet össze az s végén található párjukkal. Az összehasonlítás elvégzéséhez viszont a gépnek az egyik darabkát el kell vinnie a másikhoz, ami legalább $n/3$ lépést jelent. A gép így összesen legalább $n/3$ bit információt szállít el legalább $n/3$ cellányira. Egy lépésben b bitet képes egy hellyel arrébb vinni. Tehát az $n/3 \cdot n/3 = n^2/9 \cdot bit \cdot cella$ fuvar teljesítéséhez legalább $n^2/(9b)$ lépés kell.

Hogy mindezeket pontossá tegyük, meg kell fogalmaznunk, *mi* is az információ, amit M szállít. Erre szolgál az átkelőnapló. Legyen c az M gép egy szalagcellája. Tekintsük M futását az $s \in I^*$ bemeneten, és jegyezzük fel sorra az M állapotát azokban a pillanatokban, amikor a fej átlépi a c cella jobb oldali határát. Az így adódó $q_1, q_2, \ldots, q_m$ sorozat a c cella *átkelőnaplója*. Az első átlépéskor (ez szükségképpen jobbra lépés) M a q_1 állapotban van, a másodiknál (ami balra lépés) q_2, stb. Mint a bizonyításban látni fogjuk, az átkelőnapló tényleg úgy vehető,

mint a c határán átmenő információ hű jegyzőkönyve. Világos, hogy az átkelőnapló $m|Q|$ bittel leírható (Q az M belső állapotainak halmaza). Az átjutó információ *mennyiségének* mérésére pedig a Kolmogorov-bonyolultság lesz alkalmas. A következő takaros érvelés Wolfgang J. Paultól származik.

A tétel bizonyítása: Tegyük fel, hogy M egyszalagos Turing-gép, melyre $L_M = L_p$. Elég igazolni, hogy az $n = 3k$ hosszúságú bemeneteken M legalább dn^2 lépést tesz, ahol $d > 0$ állandó, és n elég nagy. Feltehetjük, hogy megálláskor M feje az első cellára mutat; ez a lépésszám duplázása árán elérhető.

Nézzük M működését egy $s = w1^k w^*$ alakú bemeneten, ahol w egy k hosszúságú összenyomhatatlan szó, és w^* jelöli a w szó fordítottját. Az s egy n hosszúságú palindróma.

Tekintsük az s középső harmadát tartalmazó cellák átkelőnaplóit. Ha ezek mindegyike több, mint $n/(10|Q|)$ bejegyzést tartalmaz, akkor M ezen a bemeneten több, mint $n/(10|Q|) \cdot n/3 = n^2/(30|Q|)$ lépést tett. Ekkor készen vagyunk; az ilyen bemeneten M tényleg nem végezhet $o(n^2)$ lépésben. Feltehető tehát, hogy van olyan c cella a középső harmadban, amelynek az átkelőnaplójában legfeljebb $n/(10|Q|)$ állapot szerepel. Megmutatjuk, hogy ez ellentmondáshoz vezet, ha n elég nagy.

Pontosabban azt igazoljuk, hogy *van olyan M' algoritmus, amely az n számból, a c cella i sorszámából és a c-hez tartozó átkelőnapló leírásából rekonstruálja a w szót.* M'-nek ez a bemenete leírható $2\log_2 n + 2 + n/10$ bittel. Ebből tehát $C_{M'}(w) \leq 2\log_2 n + 2 + n/10$ és az invariancia-tétel alapján $C(w) \leq 2\log_2 n + 2 + n/10 + c_{M'}$ következne. Kellően nagy n-re ez tényleg ütközik a $C(w) \geq k = n/3$ feltevéssel.

Elegendő ezután a dőlt betűs állítást bizonyítani. Az M' algoritmus sorra veszi a k hosszúságú $x \in I^*$ szavakat. Egy ilyen szóval a következőket teszi:

(1) Felírja M szalagjának elejére az $x1^{i-k}$ szót.

(2) Szimulálja M lépéseit, amíg az nem lép túl a c cellán.

(3) Ha M megáll elutasító állapotban, akkor M' az (1)-től újrakezdi a következő x szóval.

(4) Ha M elfogadó állapotban áll meg, akkor M' kiírja az x szót, és megáll.

(5) Ha M feje jobbra el akarja hagyni c-t, akkor ellenőrzi, hogy M belső állapota megegyezik-e az átkelőnaplóban következővel (ez kezdetben q_1). Ha nem, akkor újra kezdi a munkát a következő x szóval.

(6) Ha az állapotok egyeznek, akkor M' átugorja M-nek a c-n túli működését. Ez annyit tesz, hogy M-et az átkelőnapló szerint következő (ez először q_2 lehet) állapotba teszi, a fejét a c cellára állítja. Ezután (2) szerint folytatja a szimulációt.

A lényeges észrevétel az, hogy ha az (5)-beli teszteknél mindig *igen* a válasz, akkor a szimulációnál az első i cellán pontosan ugyanaz történik, mint ami akkor

történne, ha M-et az $x1^k w^*$ szóval indítanánk el. Ez közvetlenül adódik a Turing-gép definíciójából: a c-től jobbra eső részen a működés csak az átlépő állapoton keresztül függ a szalag elején történtektől. Ugyanígy a szalag elején a c-től jobbra levő munka csak a visszatérő állapoton keresztül tud hatni.

Az $x = w$ szó nyilván minden tesztet túlél. M éppen a palindrómákat fogadja el, ezért a szimuláció pontosan akkor áll meg M elfogadó állapotánál, ha $x = w$. Az M' algoritmus tehát tényleg előállítja a w szót. Ezzel a bizonyítás teljes. □

3. Véletlen sorozatok

A Kamra színházban vagyunk. Kezdődik a *Rosencrantz és Guildenstern halott*. Kihunynak a fények, minden sötét. Valami nesz hallható, aztán valaki – később megtudjuk, Rosencrantz az – azt mondja, hogy *fej*. Majd megint nesz, és megint *fej*. Ez többször ismétlődik. Egy idő után a közönség felismeri az „algoritmust", enyhe derültség, és az egyik pénzkoppanás után már a nézők is súgják, hogy *fej*. A publikum érzékeli, hogy nagyon nem szokványos, ami történik.

Ha pénzdarabot dobálunk fel, és mondjuk ötvenszer egymás után *fej* adódik, mint ahogy ez Stoppard darabjában történik, akkor úgy érezzük, hogy ez igen különös, és egyáltalán nem véletlenszerű. Egy véletlen sorozatot kuszának, szabályosságok nélkülinek képzelünk. A valószínűségszámítás jól ismert alapfogalmai viszont nem adnak módot ilyesfajta különbségtételre. A csupa fej sorozat ugyanolyannak látszik, mint bármelyik másik sorozat, hiszen minden egyes sorozat ugyanazzal a 2^{-50} eséllyel fordul elő.

A Kolmogorov-bonyolultság talán legérdekesebb vonása, hogy segítségével értelmesen definiálható a véletlen sorozat. A kapott fogalom igen határozottan megkülönbözteti a szabályos sorozatokat a kuszáktól. A többféle lehetséges meghatározás közül itt csak egyet ismertetünk. Jelölje I^∞ a végtelen 0-1 sorozatok halmazát. Egy $x \in I^\infty$ sorozatra legyen x_n az x első n bitjéből álló szó.

Definíció (Kolmogorov-véletlen sorozat): *Az $x \in I^\infty$ sorozat egy* véletlen sorozat, *ha* $\lim_{n\to\infty} \frac{C(x_n)}{n} = 1$.

Egy sorozatot tehát akkor tekintünk véletlennek, ha a kezdőszeletei véges sok kivétellel nagy Kolmogorov-bonyolultságúak (közel összenyomhatatlanok). Nevezzünk egy $x \in I^\infty$ sorozatot algoritmussal előállíthatónak, ha van olyan Turing-gép, amely a binárisan megadott n bemeneten éppen az x_n szót adja eredményül.

Állítás: *Ha az $x \in I^\infty$ sorozat algoritmussal előállítható, akkor x nem véletlen sorozat.*

Bizonyítás: Legyen M egy az x-et előállító Turing-gép. Ekkor $C_M(x_n) \leq \log_2 n + 1$. Az invariancia-tétel szerint $C(x_n) \leq \log_2 n + 1 + c_M$, amiből $\lim_{n\to\infty} \frac{C(x_n)}{n} = 0$. □

A „csupa fej" sorozat tehát nem véletlen sorozat. A fogalom sok más értelemben is megfelel a véletlenről meglevő intuitív képünknek. Igazolható például, hogy ha x egy véletlen sorozat, akkor x_n-ben körülbelül ugyanannyi nulla van, mint egyes.

A véletlen sorozat definíciójával kapcsolatban nem árt némi óvatosság. Voltak olyan nevezetes kísérletek, amelyekről később kiderült, hogy túl szigorúak abban az értelemben, hogy egyetlen sorozat sem teljesíti az előírt kikötéseket. Itt most nem ez a helyzet. Sőt, az is igaz, hogy I^∞ majdnem minden eleme véletlen sorozat.

Tétel: *Tegyük fel, hogy az $x \in I^\infty$ sorozat bitjeit egymástól függetlenül $1/2$ valószínűséggel választjuk. Ekkor x 1 valószínűséggel véletlen sorozat lesz.*

Bizonyítás: A $C(x_n) \leq n + c$ egyenlőtlenségek miatt ha x nem véletlen, akkor van olyan $\epsilon > 0$, hogy $C(x_n)/n < 1-\epsilon$ teljesül végtelen sok n-re. Legyen $k \in \mathbb{Z}^+$ és

$$S_k = \{x \in I^\infty;\ \text{van végtelen sok } n\text{, melyre } C(x_n)/n < 1 - 1/k\}.$$

Ha x nem véletlen, akkor az előzőek szerint $x \in S_k$ alkalmas k-ra. Innen

$$Prob(x \text{ nem véletlen}) \leq \sum_{k=1}^{\infty} Prob(x \in S_k).$$

Elég tehát megmutatni, hogy a $Prob(x \in S_k)$ valószínűségek mind nullák. Legyen ezután k egy rögzített pozitív egész, és jelölje A_n azt az eseményt, hogy $C(x_n)/n < 1 - 1/k$. Világos, hogy $x \in S_k$ egyenértékű azzal, hogy az A_n események közül végtelen sok következik be. A Borel–Cantelli-lemma [6] alapján elég belátni, hogy a $\sum_{n=1}^{\infty} Prob(A_n)$ sor konvergens. A $Prob(A_n)$ mennyiség becsléséhez először megjegyezzük, hogy a korábbiak szerint kevesebb, mint $2^{(1-1/k)n+1}$ szó Kolmogorov-bonyolultsága lehet kisebb, mint $(1 - 1/k)n$. Ezt használva

$$Prob(A_n) < \frac{2^{(1-1/k)n+1}}{2^n} = 2 \cdot 2^{(1-1/k)n-n} = 2(2^{-1/k})^n = 2(1/\sqrt[k]{2})^n.$$

A $\sum_{n=1}^{\infty} Prob(A_n)$ sor majorálható egy konvergens geometriai sor kétszeresével. A bizonyítás ezzel teljes. □

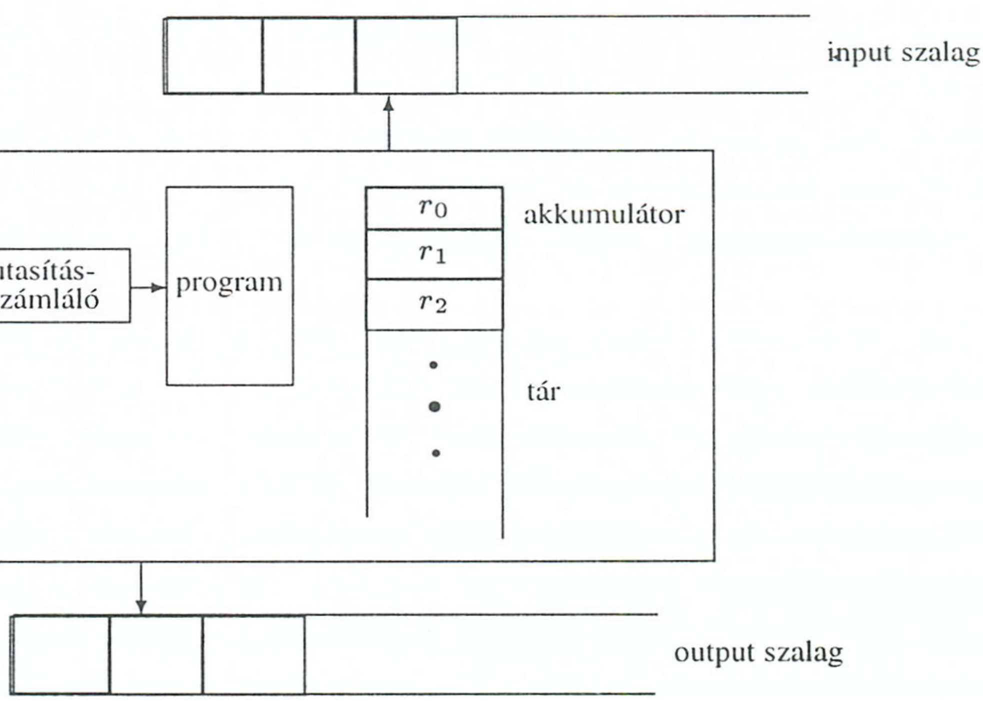
[6] A Borel–Cantelli-lemma szerint ha $A_1, A_2, \ldots$ olyan események egy valószínűségi mezőből, amelyekre $\sum_{n=1}^{\infty} Prob(A_n)$ konvergens, akkor 0 a valószínűsége annak a B eseménynek, hogy az A_i események közül végtelen sok bekövetkezik. A lemma azonnal adódik abból a tényből, hogy tetszőleges n-re a B benne van az $A_n, A_{n+1}, \ldots$ események egyesítésében, tehát $Prob(B) \leq \sum_{l=n}^{\infty} Prob(A_l)$, és a jobb oldali összeg n választásával tetszőlegesen kicsivé tehető.

7.10. A közvetlen elérésű gép (RAM)

Itt egy olyan gépmodellt tárgyalunk, mely a Turing-gépeknél sokkal jobban hasonlít a szokásos számítógépekre. A RAM rövidítés az angol *random access machine* elnevezés kezdőbetűiből származik. A gép nevében a *közvetlen elérés*, illetve a *random access* arra utal, hogy a memóriája a tömbökhöz hasonlóan egyetlen elemi lépéssel elérhető cellákból áll. A modell bevezetésével több célunk is van. Először is: a közvetlen elérésű gép a Turing-gépeknél sokkal kényelmesebb eszköz, ha algoritmusokat akarunk tervezni. Másfelől segítségével pontosan, ugyanakkor eléggé valósághűen definiálható a számítások időköltsége. Végezetül pedig a modell fontos adalékot szolgáltat a Church–Turing-tézishez. Megmutatjuk, hogy a közvetlen elérésű gépek szimulálhatók Turing-gépekkel.

A közvetlen elérésű gép alkotórészei a következők: egy kizárólag olvasásra használható *input szalag*, egy csak írható *output szalag*, a *belső tár* és a *programtár*. A szalagok és a belső tár cellákból állnak és (egyirányban) végtelen hosszúak. Egy cella egy tetszőleges egész számot tartalmazhat. A belső tár celláit a természetes számokkal sorszámozottaknak tekintjük. Kitüntetett a nulladik cella, az *akkumulátor*, ami az elemi műveletek egyik operandusát, és általában az eredményét is tartalmazza.

A programtárban található a gép *programja*. A program sorszámozott – szokásos kifejezéssel: *címkézett* – *utasítások* véges sorozata. Az *utasításszámláló* egy mutató, ami az éppen végrehajtandó utasításra mutat. A gép – értelemszerűen – az input szalagról veszi a számítás bemenő adatait; az eredmény pedig az output szalagon jelenik meg.



A következő táblázat a közvetlen elérésű gép utasításait tartalmazza. Ez egyike csupán a lehetséges és értelmes utasításkészleteknek. Az irodalomban többféle változattal találkozhat az olvasó. A táblázatban az utasítás neve után az utasítás paramétere szerepel.

Aritmetika		*Adatmozgatás*		*Vezérlés*	
ADD	*op*	LOAD	*op*	JUMP	*címke*
SUB	*op*	STORE	*op*	JGTZ	*címke*
MULT	*op*	READ	*op*	JZERO	*címke*
DIV	*op*	WRITE	*op*	HALT	

Itt *op* az adott utasítás (egyik) operandusa. Az operandusok háromfélék lehetnek. Jelöljön i egy egész számot. Az $= i$ alakú operandus – az ún. közvetlen operandus – az i egészet jelenti. Az i operandus a belső tár i-edik cellájának a tartalmát jelenti. A $*i$ formájú operandus az indirekt címzés eszköze. Jelentése az i-edik tárcellában található sorszámmal azonosított cella tartalma. Tehát ha például $r[0] = -4$ és $r[1] = 0$, akkor a $*1$ operandus jelentése -4. Az i és $*i$ alakú hivatkozásokban i szükségképpen nemnegatív egész. (A hibák kezelésével kapcsolatos kérdésektől eltekintünk.)

Az aritmetikai műveletek az összeadás (ADD), kivonás (SUB), szorzás (MULT) és osztás (DIV) első argumentuma az akkumulátor, a másodikat az *op* paraméter adja. A művelet eredménye az akkumulátorba kerül. Például ha $r[0] = 17$ és $r[1] = 2$, akkor DIV 1 hatására a $8 = \lfloor 17/2 \rfloor$ érték kerül az akkumulátorba.

A READ *op* utasítás hatására az aktuális input cella tartalma az *op* paraméter által leírt tárcellába kerül. Az input szalagon az utasítás végrehajtása után a fej egy cellányit jobbra lép. A WRITE *op* az aktuális output cellába írja az *op*-pal azonosított egészet. Itt *op* közvetlen operandus is lehet. A fej az írás után egy hellyel jobbra lép az output szalagon. A STORE *op* utasítás hatására az akkumulátor tartalma az operandus által megadott (esetleg indirekt) című tárcellába kerül; közvetlen operandus itt nem használható, mert nem jelöl címet. A LOAD *op* utasítás a fordított irányú adatmozgatásra szolgál; itt megengedett a közvetlen (konstans) operandus is.

A HALT utasítás hatására a program megáll. A további három vezérlést szabályozó utasítás ugró utasítás. Ha az akkumulátor tartalmára teljesül az ugrási feltétel, akkor a program a *címke* által megadott sorszámú utasítással folytatódik. Minden más esetben a programtár következő utasításával folytatódik a munka. Az ugrási feltételek a következők: JZERO: $r[0] = 0$, JGTZ: $r[0] \geq 0$, JUMP: üres – azaz mindig teljesül.

A RAM-modell két értelemben idealizálja a szokásos egyprocesszoros architektúrákat: nincs megkötés a tár, illetve a szalagok méretére. Ezenfelül nincs „szó-

hossz", azaz egy cella tetszőlegesen széles egész számot tartalmazhat. Egyebekben a modell megfelel az assemblerben programozható és némi perifériával kiegészített processzornak.

Költségszámítás

A RAM-programok időigényének mérésére kétféle mérőszámot szokás alkalmazni. Az egyik az *uniform költség*. Egy program uniform költsége a futása során végrehajtott utasítások száma. Úgy is mondhatjuk, hogy minden elemi utasítás költsége 1. A rendezés, a keresés és a gráfalgoritmusok kapcsán lényegében ezt a mérési módot használtuk. Becsléseink a módszerek uniform költségére adtak nagyságrendi korlátokat.

Az uniform költség használatakor nem árt egy kis körültekintés. Mindenféle huncutságot el lehet követni úgy, hogy óriási méretű számokkal dolgozunk. Így alacsony költséget kaphatunk olyan számításokra is, amelyek gyakorlati megvalósítása igen drága. Egyszerű példaként tekintsük az $f(n) = 3^{2^n}$ függvényt. Az $f(n)$ érték kiszámítható lineáris, azaz $O(n)$ uniform költséggel: legyen $x := 3$, majd n-szer iterálva $x := x^2$. Ugyanakkor vegyük észre, hogy a k-adik iterációs lépésben két 2^k-jegyű számot szorzunk össze. Az eredménynek tehát 2^{n+1} jegye lesz. Ha mondjuk $n = 1000$, akkor 1000 elemi műveletet végzünk, a 2^{1001} bitből álló eredmény leírásához viszont a világ minden papírja sem elegendő.

A *logaritmikus költség* kiküszöböli ezt a fogyatékosságot, amennyiben érzékeny a számításban szereplő adatok méretére is. Ennél a számolási módnál egy utasítás költsége a benne szereplő adatok összhossza. Egy egész szám hossza a bináris jegyeinek száma $+1$, hogy az előjelre is tekintettel legyünk. Az adatok nélküli utasítások (JUMP, HALT) költsége 1. Egy program logaritmikus költsége a végrehajtott utasítások költségeinek az összege.

Példa: ADD $*1$ uniform költsége 1. A logaritmikus költsége viszont

$$\text{hossz}(r[0]) + \text{hossz}(r[1]) + \text{hossz}(r[r[1]]) + \text{hossz}(1),$$

ahol $r[i]$ a belső tár i-edik cellájának a tartalmát jelöli.

A logaritmikus költség a valósághoz hű mérőszámot ad olyan programok esetén, amelyek nem tartalmaznak MULT, illetve DIV utasításokat. Ennek az alapja az, hogy a többi elemi művelet ténylegesen megvalósítható a benne szereplő adatok hosszával arányos időköltséggel. Például az összeadás és a kivonás esetén az iskolában tanult táblázatkitöltő módszer ilyen tulajdonságú. A szorzásra és az osztásra viszont mindeddig nem találtak olyan algoritmust, ami n bites bemenetek

esetén n-nel arányos számú bit-művelettel megadná az eredményt[7].

A két költségfogalmat összevetve megállapíthatjuk, hogy az uniform költséget könnyebb számolni, becsülni, a logaritmikus költség pedig pontosabb, valóságosabb mérőszámot ad. Az uniform költséget akkor értelmes használni, ha tudunk valami korlátot a számítás során keletkező adatok méretére. Ha egy RAM-program végig legfeljebb l hosszúságú adatokkal dolgozik, és az uniform költsége m, akkor a logaritmikus költsége $O(lm)$.

Az igazi számítógépek processzorai szó-szervezésűek. Ez egyebek közt azt jelenti, hogy a szóhossz korlátozza az elemi műveletek operandusainak hosszát. Másfelől az alkalmazások tekintélyes részénél a természetesen adódó elemi adatok és az ezekből nyert részeredmények belefénnek egy gépi szóba. Az ilyen esetekben az uniform költség, vagyis az elemi műveletek száma használható mérőszámot ad.

Szimulációk

A következő tétel azt állítja, hogy a közvetlen elérésű gépek és a Turing-gépek számító ereje megegyezik. Ezzel adalékot szolgáltat a Church–Turing-tézishez. Másfelől a RAM-ok csiszoltabb architektúrájuknak köszönhetően gyorsabbak. A hatékonyságbeli nyereség azonban egy négyzetes korláton belül marad. A tétel lehetővé teszi, hogy ha hatékony algoritmust akarunk készíteni, akkor a programozás természetéhez közelebb álló RAM-modellt használjuk.

Tétel (Turing-gép↔RAM szimulációk):
(1) Tetszőleges M Turing-gép szimulálható $O(T_M(n)\log T_M(n))$ logaritmikus költségű RAM programmal. A szimuláció uniform költsége $O(T_M(n))$.
(2) Egy $t(n)$ logaritmikus költségű, MULT és DIV utasításokat nem tartalmazó RAM-program szimulálható olyan N Turing-géppel, melynek az időigényére $T_N(n) = O(t^2(n))$ teljesül.

Bizonyítás: (vázlat)
(1) Legyen M egy k-szalagos Turing-gép. Az általánosság rovása nélkül feltehetjük, hogy M szalagjelei és belső állapotai is természetes számok. Feltehetjük azt is, hogy az elfogadás/elutasítás tényét M az output szalagjának első mezején jelzi. A szimuláció megkezdése előtt M bemenete a RAM input szalagjára van írva. Egy cellában egyetlen szalagjel szerepel. A RAM belső tárának első c celláját munkaterületként használjuk. A c egy az M-től függő állandó. Ebben az első részben tároljuk az M aktuális belső állapotát, és itt kap helyet az a k cella is, amelyekben az M fejeinek helyzetét ábrázoljuk.

[7] A számítógépes aritmetika talán legnevezetesebb nyitott kérdése, hogy van-e ilyen algoritmus. A ma ismert leggyorsabb módszert Arnold Schönhage és Volker Strassen találta. Az algoritmus két n jegyű szám szorzásához $O(n\log n\log\log n)$ bit-műveletet használ, és nem praktikus.

A belső tár további részében *összefésülve*[8] tároljuk az M szalagjainak tartalmát: az M j-edik szalagjának i-edik celláját a belső tár $c-1+j+ki$ sorszámú cellája jelenti.

A RAM programja tartalmazza az M átmeneteinek leírását. Az M egy lépését a közvetlen elérésű gép a bemenet hosszától független konstans számú lépésben utánozza. A fejezet elején vázolt **if...then...** utasítások megvalósíthatók konstans sok RAM-alapművelettel. A RAM az M celláinak megfelelő helyeket indirekt címzéssel éri el a tár első részében levő mutatók alapján.

A programnak n jelből álló bemenet esetén szimulálnia kell az M legfeljebb $T_M(n)$ lépését. Ennek az uniform költsége $O(T_M(n))$. Ugyanezen a korláton belül maradva átmásolhatjuk az M output szalagjának (érdemi) tartalmát a RAM output szalagjára. A szimuláció uniform költsége tehát összesen is $O(T_M(n))$. A logaritmikus költség becsléséhez nézzük a szimuláció során fellépő számok méretét: az M szalagjelei és belső állapotai (M-től függő) konstans hosszúságúak, a fejek helyét leíró mutatók pedig $O(\log T_M(n))$ bittel ábrázolhatók. A szimuláció logaritmikus költsége tehát $O(T_M(n)\log T_M(n))$.

(2) A RAM-programot szimuláló N gép szalagjelei legyenek $\{0, 1, +, -, \#, \ddot{u}\}$. Az első négy jellel binárisan írt előjeles egészeket kódolunk, a $\#$ elválasztójelként fog szolgálni. N-nek öt szalagja lesz. Az elsőn a RAM belső tárát ábrázoljuk, a másodikon az akkumulátor tartalmát. A harmadik szalag munkaszalag lesz, a negyedik és az ötödik pedig a RAM input, illetve output szalagjának felel meg. Az utóbbi két szalagon a binárisan ábrázolt egészeket $\#$ jelek választják el egymástól. N szalagjai tehát így néznek ki.

A RAM belső tára

Az akkumulátor tartalma

Munkaszalag

A RAM input szalagja

A RAM output szalagja

Külön figyelmet érdemel az első szalag, amin a RAM belső memóriáját szimuláljuk. Ez a szalag *cím# adat* alakú feljegyzéseket tartalmaz, ## jelekkel elválasztva.

$\#\#\text{cím}_1\#\text{adat}_1\#\#\text{cím}_2\#\text{adat}_2\#\#\text{cím}_3\#\text{adat}_3\#\#\text{cím}_1\#\text{új-adat}_1\#\#\text{cím}_3\#\text{új-adat}_3\#\#\text{cím}_2\#\text{új-adat}_2 \ldots\ldots$

[8] Ezt az ötletet érdemes megjegyezni. Változatait gyakran használják az ún. *dinamikus allokációt* igénylő helyzetekben: amikor is egyetlen tartományban több, dinamikusan változó méretű állományt kell kezelni.

Például az $110\# - 10001$ pár azt jelenti, hogy $r[6] = -17$. Egy adott című rekesz „beolvasását" N úgy végzi, hogy végigpásztázza az első szalagot a legutolsó olyan feljegyzésig, amelynek a *cím* része egyezik az adott címmel. A megfelelő adatrészt ezután az akkumulátort ábrázoló szalag elejére írja, és a végére üresjelet tesz. A „kiírás" úgy történik, hogy a megfelelő *cím#adat* párt az első szalag végére írjuk. Azért használunk mindig új helyet, mert előfordulhat, hogy a szóban forgó cellában levő szám mérete megnőtt, és már nem férne el a korábbi helyére.

A RAM alaputasításainak az N állapotcsoportjai felelnek meg. Például van egy az összeadáshoz tartozó állapotcsoport. Amíg N az összeadást végzi, addig az aktuális belső állapota ebből a csoportból kerül ki. A következő RAM-utasításra lépést, illetve az ugrásokat N az állapotcsoportok közötti átmenettel valósítja meg.

Az alaputasítások közül a LOAD és a STORE szimulációját már vázoltuk. További példaként tekintsük az ADD $*9$ megvalósítását:
1. Először N megkeresi az első szalagon a legutolsó olyan feljegyzést, aminek a címrésze $\#\#1001\#$. Ha nincs ilyen, akkor N megáll, a szimuláció nem folytatható, hiszen az indirekt címzés értelmetlen. Ha a keresett cím szerepel a szalagon, akkor a mögötte levő a adatot N a harmadik szalag elejére másolja.
2. Megkeresi az első szalagon az utolsó feljegyzést, aminek a címrésze a harmadik szalagon levő a érték. Ha ez nem lehetséges, akkor N megáll. Ha a keresés sikeres volt, akkor az a címhez tartozó b adatrészt a harmadik szalag elejére másolja.
3. A harmadik szalagon található b számot hozzáadja az akkumulátor tartalmához, vagyis a második szalagon levő számhoz; az eredmény a második szalagon jelenik meg.

Ezeket a mintákat követve nem nehéz meggondolni, hogy az alaputasítások – a MULT és DIV kivételével – megvalósíthatók úgy, hogy N lépésszáma az utasításban szereplő adatok hosszával plusz az első szalag érdemi részének hosszával arányos legyen. Ha a RAM bemenetének mérete n volt, akkor a logaritmikus költség definíciója szerint ez az összhossz $O(t(n))$, így egy lépés szimulációjának a költsége is $O(t(n))$.

A RAM $t(n)$ lépésének szimulációjához tehát N legfeljebb $t(n)O(t(n)) = O(t^2(n))$ lépést tesz. □

Feladat: Mutassuk meg, hogy két n jegyű egész szorzása, illetve osztása Turing-géppel $O(n^2)$ lépésben elvégezhető.

Feladat: Mutassuk meg, hogy egy tetszőleges (MULT és DIV utasításokat is használó) RAM-program szimulálható egy olyan N Turing-géppel, amelyre $T_N(n) = O(t^3(n))$.

A tétel, illetve az utóbbi feladat szerint ha egy algoritmikus probléma megoldható $T(n)$ költséggel az egyik modellben, akkor megoldható legfeljebb $O(T^3(n))$

költséggel a másik modellben is. A két modell tehát *polinomiálisan összehasonlítható* abban az értelemben, hogy az egyiknél számított költség becsülhető a másikon vett költség egy polinomjával. Ha tehát pusztán az érdekel bennünket, hogy egy feladatra van-e $O(n^c)$ költségű algoritmus valamilyen $c > 0$ kitevővel[9], akkor közömbös, hogy melyik gépmodellben gondolkodunk.

[9]Hatékony algoritmusok keresésekor gyakran ez az első kérdés, amivel foglalkozunk.

8.

Az NP nyelvosztály

Az ember s az idő mindig összetalálnak,
mint a keserűlapi saját árnyékával.
TAMÁSI ÁRON

Ebben a fejezetben először algoritmusok időbeli és tárhasználat szerinti hatékonyságát vizsgáljuk. Definiálunk néhány olyan jellemzőt, melyek lehetővé teszik a hatékonyság pontosabb értelmezését. E fogalmak segítségével megadható a (legalábbis elméleti értelemben) hatékonyan kezelhető algoritmikus problémák köre (P és FP osztályok). A legtöbb eddig tárgyalt feladat (nyelv, függvény) ezen osztályok egyikébe tartozik. Mint látni fogjuk, sok gyakorlati szempontból fontos feladat – a mai tudásunk szerint – kívül esik ezeken az osztályokon, vagyis nem oldható meg hatékony algoritmusokkal. E nehezebb feladatok közül kiemelkedő fontosságúak azok, amelyek az NP osztályba tartoznak. Itt most csak azt emeljük ki, hogy nagyon sok fontos algoritmikus feladat vezet NP-beli problémához. Az NP-beli nyelvekkel kapcsolatban első közelítésként a bevezető példáink egyikére, Arthur király kérésére utalunk. Képzeljük el, hogy a király csak annyit kérdez Merlintől, hogy párokba állíthatók-e a lovagok és az udvarhölgyek úgy, hogy a vonzalmakat is figyelembe vegyük. Merlin a válasz mellé egyszerűen ellenőrizhető bizonyítékot mellékelhet: a megfelelő párok listáját. Ennek birtokában a király hatékonyan ellenőrizheti a válasz helyességét. Az NP-beli nyelvek éppen ezzel a tulajdonsággal jellemezhetők. Az *igen* válasznak van gyorsan ellenőrizhető bizonyítéka.

Külön figyelmet fogunk szentelni az NP osztály legnehezebb feladatainak, az NP-teljes feladatoknak. Az NP-osztályra úgy gondolhatunk mint a természetesen felmerülő nyelvek nagy gyűjtőhelyére; az alján vannak a hatékonyan, gyorsan felismerhető nyelvek (a P osztály), a tetején pedig a nehéz, gyors algoritmussal nem megoldható problémák, az ún. NP-teljes nyelvek. Utóbbiakkal azért foglalko-

zunk behatóbban, mert szinte minden alkalmazási területen előfordulnak. Az NP-teljességgel kapcsolatos ismeretek gyakran segítenek annak megítélésében, hogy egy elénk kerülő algoritmikus feladat megoldására remélhetünk-e gyors módszert.

8.1. Idő- és tárkorlátok

Kétmilliárd férfi húszezer évet borotválkozik naponta. GARACZI LÁSZLÓ

Szeretnénk viszonylag pontos fogalmakat adni arra, hogy egy algoritmus gyors, illetve hogy hatékonyan bánik a tárral. Ezt úgy tesszük, hogy korlátozzuk az algoritmus (Turing-gép) számolási idejét vagy a felhasznált tárcellák számát. Nyilvánvalóan nem lenne jó abszolút, a bemenettől független korlátokat bevezetni, hiszen természetesnek tartjuk, hogy hosszabb inputon egy módszer több időt/tárat használ; már pusztán a bemenet elolvasása, értelmezése több munkát jelent. A két törekvés összebékítésének egy lehetséges módja, hogy az idő- és tárfelhasználást az *input hosszának függvényében* vizsgáljuk. Legyen $t : \mathbb{Z}^+ \to \mathbb{Z}^+$ egy függvény, melyre minden $n \in \mathbb{Z}^+$ esetén $t(n) \geq n$ teljesül.

Definíció ($t(n)$ időkorlátos TG):
Az M Turing-gép $t(n)$ időkorlátos, ha n hosszú inputokon legfeljebb $t(n)$ lépést tesz (más szóval $T_M(n) \leq t(n)$).

A $t(n)$ függvény tehát korlátozást ír elő az M gép lépéseinek a számára. A $t(n) \geq n$ azt a természetes feltevést tartalmazza, hogy a gép legalább végigolvashatja az inputot a megadott időkorláton belül maradva. Az M algoritmust (TG-t) akkor tekinthetjük gyorsnak, ha $t(n)$ egy lassan növekedő függvény.

A 7.1. pontbeli első példa, a hárommal való oszthatóságot ellenőrző M Turing-gép n hosszú inputokon legfeljebb $n+1$ lépést tesz, tehát mondhatjuk, hogy $n+1$ időkorlátos. A másik példa, az $f(n) := n+1$ függvényt kiszámító M gép nem lesz $t(n)$ időkorlátos semmilyen t függvénnyel, mert vannak olyan inputok, amiken a gép sohasem áll meg.

Feladat: Adjunk meg egy olyan $n+1$ időkorlátos N Turing-gépet, melyre $L_N = L_M$ és $f_N = f_M$, ahol M a fenti TG.

A hatékony algoritmus fogalmának a birtokában megpróbálkozhatunk a feladatok hatékonyság szerinti osztályozásával. Azokat a feladatokat (nyelveket, függvényeket) tekintjük alacsony bonyolultságúnak, melyek megoldására létezik gyors algoritmus. Ezt készíti elő a következő fogalom.

Definíció:

$$TIME(t(n)) := \left\{ L \subseteq I^* \;\middle|\; \begin{array}{l} L \textit{ felismerhető egy } O(t(n)) \textit{ időkorlátos} \\ M \textit{ Turing-géppel} \end{array} \right\}.$$

A $TIME(t(n))$ nyelvosztályba tehát azok az L nyelvek tartoznak, amelyekhez létezik $ct(n)$ időkorlátos TG. A c állandó függhet L-től. A definíció lényeges eleme, hogy n hosszú x inputokon a számítás *mindig befejeződik* legfeljebb $ct(n)$ lépésben, tekintet nélkül arra, hogy $x \in L$ igaz-e. Ennek következményeként $TIME(t(n))$ *rekurzív nyelvekből áll.*

A legegyszerűbb példát a *lineáris időben* felismerhető nyelvek jelentik:

Példa: $TIME(n) = \{$*az* $O(n)$, *azaz lineáris időben felismerhető nyelvek*$\}$.

Algoritmikus időigény szempontjából a lineáris időben felismerhető nyelvek tekinthetők a legkönnyebbeknek. Szokásos még a $TIME(n^2)$ nyelvosztályt a négyzetes, $TIME(n^3)$-t pedig a köbös időben felismerhető nyelvek összességének nevezni. A következő nyelvosztály központi szerepet játszik a számítások elméletében:

Definíció: $\mathrm{P} = \cup_{k \geq 1} TIME(n^k)$, *a polinom időben felismerhető nyelvek osztálya.*

Példa: $L = \{0^n 1^n |\ n \geq 1\} \in TIME(n) \subseteq \mathrm{P}$. Könnyen szerkeszthető olyan kétszalagos Turing-gép, mely a fenti nyelvet lineáris időben ismeri fel.

Állítás: *Ha az* L *nyelv* $n^{\log n}$*-nél rövidebb időben nem ismerhető fel, akkor* $L \notin \mathrm{P}$.

Bizonyítás: Indirekt érveléssel tegyük fel, hogy $L \in \mathrm{P}$. Ekkor a P definíciója szerint van olyan $k > 0$, melyre $L \in TIME(n^k)$, és így alkalmas $c > 0$ konstansra $n^{\log n} \leq cn^k$ teljesülne végtelen sok n-re. Ez az egyenlőtlenség viszont ellentmondást jelent, hiszen a bal oldalon álló függvény gyorsabban nő, mint a jobboldali. □

Az időhöz hasonlóan kezelhetjük az algoritmusok tárfelhasználását és a nyelvek felismerésének tárigényét. Legyen $s : \mathbb{Z}^+ \to \mathbb{Z}^+$ olyan függvény, melyre minden $n \in \mathbb{Z}^+$ számmal igaz, hogy $s(n) \geq \log_2 n$.

Definíció ($s(n)$ tárkorlátos TG):
Az M *Turing-gép* $s(n)$ *tárkorlátos, ha* n *hosszú inputokon legfeljebb* $s(n)$ *tárcellát használ a* munkaszalagokon *(azaz* $S_M(n) \leq s(n)$*).*

Az $s(n) \geq \log_2 n$ kikötés enyhe és értelmes feltevés. Ennyi hely kell ugyanis ahhoz, hogy egy n cellából álló szalagrészt – például az input szalag érdemi részét – címezni tudjunk. Emlékeztetünk még itt arra, hogy ha M-nek csak egy szalagja van, akkor az a definíció szempontjából munkaszalagnak tekintendő. Az időosztályokkal analóg módon kapjuk a tárosztályok definícióját:

Definíció:

$$SPACE(s(n)) := \left\{ L \subseteq I^* \;\middle|\; \begin{array}{l} \textit{az } L \textit{ felismerhető egy } O(s(n)) \textit{ tárkorlátos} \\ M \textit{ Turing-géppel} \end{array} \right\}.$$

Példa: $SPACE(\log n)$ a *logaritmikus tárban felismerhető nyelvek* osztálya. Ha az L nyelv a $SPACE(\log n)$ osztályban van, akkor felismerhető egy olyan M TG-vel, ami n hosszú inputokon legfeljebb $c \log n$ tárcellát használ a munkaszalagjain. A c állandó itt is függhet L-től.

A nyelvekhez hasonló módon kaphatunk idő-, illetve tárkorlátokkal meghatározott függvényosztályokat. Csupán annyit kell tennünk, hogy a definíciókban a nyelveket felismerő TG-k helyett függvényeket kiszámoló TG-ket szerepeltetünk.

Definíció: $FTIME(t(n)) :=$ *az* $O(t(n))$ *időkorlátos TG-k által kiszámítható* $f : I^* \to I^*$ *függvények osztálya.*

Definíció: $FSPACE(s(n)) :=$ *az* $O(s(n))$ *tárkorlátos TG-k által kiszámítható* $f : I^* \to I^*$ *(parciális) függvények osztálya.*

Igen fontos osztály a P-vel analóg függvényosztály FP, a *polinom időben kiszámítható függvények osztálya*:

Definíció: FP $:= \cup_{k \geq 1} FTIME(n^k)$.

8.2. Tár–idő-tétel, nevezetes nyelvosztályok

Jóllehet a világot alkotó atomok száma mérhetetlenül nagy, nem végtelen, ezért csupán véges számú (még ha mérhetetlenül nagy is ez a szám) permutációt adhat. Ha végtelen az idő, a lehetséges permutációk száma egyszer kimeríthető, s akkor szükségszerűen megismétlődik a világegyetem. Ismét megszületsz majd egy anyaméhből, ismét kifejlődik a csontvázad, ismét ugyanebbe a kezedbe kerül ez a lap, ismét végigéled életed minden óráját a hihetetlen halál percéig. JORGE LUIS BORGES

(Így foglalja össze a nietzschei Örök Visszatérést.)

A következő állítás egy alapvető összefüggést rögzít a tár-, illetve időkorlátokkal definiált osztályok között. Ha egy nyelv felismerésére van tárkorlátos algoritmus, akkor van időkorlátos is. Az adódó időkorlát a tárkorlát exponenciális függvényével becsülhető. A gondolatmenet kezdőpontja az, hogy ha egy tárkorlátos számítás elég sokáig tart, akkor szükségképpen végtelen ciklusban van. A végtelen ciklusok felismerése, kezelése jelenti a probléma – és a bizonyításban szereplő konstrukció – érdemi részét.

Tétel (tár–idő-tétel): *Ha* $L \in SPACE(s(n))$, *akkor van olyan* L*-től függő* c *konstans, mellyel* $L \in TIME(c^{s(n)})$ *teljesül.*

Bizonyítás: Legyen M egy $S(n) = c_1 s(n)$ tárkorlátos ($c_1 \geq 1$) k-szalagos TG, mely felismeri L-et. Az M-ből kiindulva egy olyan $O(c^{S(n)})$-időkorlátos N TG-t fogunk konstruálni, melynek a nyelve szintén L. Nézzük M számításait az n hosszúságú inputokon. Egy ilyen számítás során a gép egy pillanatnyi helyzetét pontosan jellemzi az input (összesen n jel), a munkaszalagok tartalma (összesen legfeljebb $S(n)$ cella), a gép aktuális belső állapota, valamint a fejek helyzete a szalagokon. Egy ezeket rögzítő „jegyzőkönyvet" pillanatnyi helyzetleírásnak (PHL) nevezünk. Az M gép átmenetei és egy PHL ismeretében pontosan meg tudjuk mondani az M következő lépését, sőt a következő PHL-eket is. Ilyen értelemben egy PHL a számítás egy pillanatát hűen rögzítő fényképnek tekinthető. Mindebből számunkra most az a fontos, hogy ha a gép futása során egy PHL ismételten előfordul, akkor a gép biztosan *végtelen ciklusban van*, a két helyzet közötti lépéssor végtelen sokszor ismétlődik. A gép tárkorlátos voltából adódik, hogy egy számítás során csak véges sok PHL lehetséges. Így ha a gép elég sokáig fut, akkor egy PHL feltétlenül ismétlődik, tehát végtelen ciklusba kerültünk.

Nézzük ezt kicsit pontosabban! Hány darab PHL lehetséges összesen, ha a gépet n hosszú inputtal indítjuk? Érvényes a következő egyszerű felső becslés

($\#PHL$ a PHL-ek száma):

$$\#PHL \leq |Q||T|^{S(n)}(n+1)S(n)^k,$$

ahol $|Q|$ az M belső állapotainak száma, $|T|$ az M szalagjeleinek száma, az $(n+1)$ tényező az input fej lehetséges helyzeteinek a száma, az $S(n)^k$ tényező pedig a többi fej lehetséges helyzeteinek a száma. Használva, hogy $S(n) \geq s(n) \geq \log_2 n$, a $c_2 := 2^{k+1}|T|$ választással

$$\#PHL \leq \text{konstans} \cdot c_2^{S(n)}$$

adódik. Jelöljük t-vel a jobboldalon álló számot. Ha egy n-hosszú x inputon a gép t lépés után sem áll meg, akkor biztosan végtelen ciklusba került, hiszen ekkor van olyan PHL, ami két különböző lépés után előfordul. Kézenfekvő volna tehát M-et t lépés után lelőni. Az a baj ezzel az ötlettel, hogy nem feltétlenül tudunk eddig elszámolni, hiszen lehet, hogy a t korlát nem rekurzív függvénye n-nek. Olyan megoldásra van szükség, amely nem épít a t ismeretére.

Ennyi előkészület után lássunk az N gép konstrukciójához! A végtelen ciklusok felismerése és kezelése céljából megduplázzuk M-et. Legyen M_1 és M_2 a két példány. Ezeket a gépeket N részeinek tekintjük. Az M_1-et elindítjuk az x inputtal. Minden egyes lépése után ideiglenesen megállítjuk; ekkor M_2-t elindítjuk x inputtal a kezdő állapotból, és működtetjük legfeljebb addig a lépésig, ahol M_1 tart (jelölje ennek a lépésnek a sorszámát l). Az l sorszámot $O(S(n))$ extra cellán tároljuk és léptetjük. Ha valamely $j < l$-re az M_2 gép j-edik lépés utáni PHL-je megegyezik M_1-ével, akkor biztosan végtelen ciklusba kerültünk (PHL ismétlődik), tehát $x \notin L$. Ekkor N megáll elutasítva x-et. Ha ilyen ismétlődés nem fordult elő, akkor meglépjük M_1 következő, $l+1$-edik lépését, $l := l+1$, és ismételjük az előző eljárást. Természetesen ha M_1 megáll elfogadva (elutasítva) x-et, akkor N is megáll elfogadva (elutasítva) x-et.

Ezzel a megoldással biztosan felismerjük a végtelen ciklusokat: valójában már az első olyan l után megállunk, amikor egy PHL ismétlődik. Az új, összetett gép működése során ezért mindig teljesül az $l \leq t$ egyenlőtlenség. Innen már megbecsülhető N erőforrás-igénye. Beszámítva a számlálók (l és j) karbantartását és a PHL-ek összehasonlításának költségét is, a maximális futási idő legfeljebb $O(t^2) = O((c_2^{S(n)})^2) = O((c_2^2)^{c_1 s(n)})$, így $c = c_2^{2c_1}$ megfelel a tétel követelményeinek. További fontos tény, hogy a tárfelhasználás sem nőtt lényegesen: az összetett gép működéséhez $O(s(n))$ tárcella elegendő a munkaszalagokon. □

A bizonyítás minden nehézség nélkül átvihető nyelvek helyett függvényekre. Egyetlen számottevő módosítás célszerű: végtelen ciklus esetén, amikor az x inputra f nem értelmezett, legyen $f(x) = *$. Az így kapott (teljes) függvény szintén az $FSPACE(s(n))$ osztályba tartozik.

Tétel: *Ha* $f \in FSPACE(s(n))$*, akkor van olyan f-től függő c konstans, mellyel* $f \in FTIME(c^{s(n)})$. □

Nézzük most, hogy mi mondható idő- és tárkorlátokkal definiált nyelvosztályok esetén a komplemens nyelvek osztályairól. Emlékeztetésül: ha X nyelvek egy osztálya, akkor a coX nyelvosztály éppen $I^* \setminus L$ alakú nyelvekből áll, ahol $L \in X$. Így pl. a co$TIME(t(n))$ nyelvosztályban a $TIME(t(n))$-beli nyelvek komplementerei vannak.

Egyszerűen adódik, hogy $TIME(t(n)) =$ co$TIME(t(n))$. Legyen ugyanis M egy $ct(n)$ időkorlátos M TG, ami az L nyelvet ismeri fel. Ha megcseréljük az M elfogadó és elutasító (azaz nem elfogadó) állapotait, akkor a kapott N TG éppen az $I^* \setminus L$ nyelvet ismeri fel, és szintén $ct(n)$ időkorlátos. Analóg állítás érvényes tárosztályokra is; ennek az igazolása viszont kevésbé egyszerű.

Tétel: $SPACE(s(n)) = coSPACE(s(n))$.

Bizonyítás: Legyen $L \in SPACE(s(n))$. Alkalmazzuk a tár–idő-tétel szimulációját. Az adódó N TG szintén $O(s(n))$ tárkorlátos, és *minden inputra megáll.* Erre a gépre már működik időosztályoknál bevált ötlet: cseréljük fel az elfogadó és az elutasító állapotokat. □

Megemlítünk itt két további fontos nyelvosztályt, az *exponenciális időben felismerhető*, valamint a *polinom tárban felismerhető* nyelvek osztályait.

Definíció: EXPTIME $:= \cup_{k \geq 1} TIME(2^{n^k})$.

Definíció: PSPACE $:= \cup_{k \geq 1} SPACE(n^k)$.

Az EXPTIME osztályra úgy nézhetünk, mint a gyakorlatban előforduló nyelvek (eldöntési feladatok) univerzumára. Ezt úgy értjük, hogy nemigen van olyan praktikus feladat, ami ezen kívül levő, még nehezebb nyelvhez vezetne. Érvényesek a következő tartalmazási viszonyok:

Tétel: P $\subseteq$ PSPACE $\subseteq$ EXPTIME

Bizonyítás: Ha M egy $t(n)$ időkorlátos TG, akkor szükségképpen $ct(n)$ tárkorlátos is, hiszen egy lépésben legfeljebb annyi új cellára léphet, mint a szalagjainak száma. Ezt alkalmazva kapjuk, hogy $TIME(n^k) \subseteq SPACE(n^k)$, amiből P $\subseteq$ PSPACE következik. A másik állítást illetően legyen $L \in$ PSPACE. Ekkor $L \in SPACE(n^k)$, valamely k-ra. A tár–idő-tétel szerint ekkor van olyan $c > 0$, hogy $L \in TIME(c^{n^k}) \subseteq TIME(2^{\lceil c \rceil n^k}) \subseteq TIME(2^{n^{k+1}}) \subseteq$ EXPTIME. □

Megjegyezzük még – a $\subset$ jellel valódi tartalmazást jelölve –, hogy $TIME(t(n)) \subset \mathcal{R}$, $SPACE(s(n)) \subset \mathcal{R}$ és EXPTIME $\subset \mathcal{R}$. Ezek a tények a korábban már megismert *átlós eljárás* alkalmazásával igazolhatók. Az ötlet érzékeltetésére mutatunk egy rekurzív nyelvet, ami nincs az EXPTIME osztályban. Legyen

$$L = \{w \in I^*;\ \text{az } M_w \text{ TG létezik, és legfeljebb } 2^{2^{|w|}} \text{ lépésben elutasítja } w\text{-t}\}.$$

Itt $|w|$ jelöli a w szó hosszát. Először gondoljuk meg, hogy ha M egy $t(n)$ időkorlátos TG, akkor végtelen sok olyan $y \in I^*$ szó van, amire M_y létezik, és ugyanúgy viselkedik, mint M. Az utóbbi azt jelenti, hogy M_y is $t(n)$ időkorlátos és ugyanazt a nyelvet ismeri fel, mint M: $L_{M_y} = L_M$. Ilyen gépeket kaphatunk például úgy, hogy M belső állapotainak halmazát bővítjük olyan állapotokkal, amelyek sosem érhetők el a kiinduló helyzeteiből. (Tulajdonképpen az történik, hogy egy programot olyan részletekkel bővítünk, amelyekre sohasem kerül a vezérlés. A program ugyanúgy működik, mint az eredeti, de a leírása hosszabb lesz.)

Most megmutatjuk, hogy L nincs benne a $TIME(2^{2^{n-1}})$ nyelvosztályban. Ebből, EXPTIME $\subseteq TIME(2^{2^{n-1}})$ miatt következik, hogy $L \notin$ EXPTIME. Indirekt gondolkodva tegyük fel, hogy L felismerhető egy $c2^{2^{n-1}}$ időkorlátos M TG-vel. Legyen n_0 olyan nagy, hogy $c2^{2^{n-1}} < 2^{2^n}$ teljesüljön, ha $n > n_0$. Legyen w egy n_0-nál hosszabb szó, melyre M_w létezik, és ugyanúgy viselkedik mint M. Ez a gép is $c2^{2^{n-1}}$ időkorlátos, és $L = L_{M_w}$. A befejezés hasonlít a diagonális nyelvnél látottakra: ha $w \in L$, akkor M_w elfogadja w-t $c2^{2^{|w|-1}} < 2^{2^{|w|}}$ lépésben, amiből L definíciója szerint $w \notin L$ következik. Ugyanígy képtelenséghez vezet a $w \notin L$ feltevés is.

Feladat: Mutassuk meg, hogy a fenti L nyelv rekurzív.

A P, FP, PSPACE, EXPTIME osztályok *robusztusak* abban az értelemben, hogy nagymértékben függetlenek a gépmodelltől, amelynek a segítségével definiáltuk őket. Például szorítkozhattunk volna csak az egyszalagos Turing-gépekre, vagy használhattuk volna a RAM modellt. A TG-RAM szimulációkra kapott korlátokból azonnal következik például, hogy

$$\mathrm{P} = \left\{ L \subseteq I^* \;\middle|\; \begin{array}{l} \text{van olyan } k \in \mathbb{Z}^+ \text{ és } c > 0, \text{ hogy } L \text{ felismerhető} \\ cn^k \text{ logaritmikus költségű RAM-programmal} \end{array} \right\}.$$

Feladat: Mutassuk meg, hogy a lineáris időben felismerhető nyelvek osztálya $TIME(n)$ nem robusztus osztály. (Nézzük a palindrómák nyelvét egy-, illetve kétszalagos gépeken.)

Elméleti szempontból a polinom idejű algoritmusokat tekinthetjük hatékonyaknak. Ez több értelemben is megfelel a hatékonyságról meglevő tapasztalati képünknek. A lineáris, kvadratikus, sőt még a köbös algoritmusok gyorsaságával is általában elégedettek vagyunk. Bizonyos esetekben az ennél valamivel nagyobb kitevőjű módszerekkel is békét tudunk kötni. A sokkal nagyobb időigényű eljárások viszont (legalábbis amelyek viszonylag természetesen merülnek fel) használhatatlanok hosszú inputokon. Nézzünk mondjuk egy 2^n időigényű módszert $n = 1000$ hosszúságú input esetén! Ez a méret nem jelent gondot a mai gépeken n, n^2 sőt n^3 lépésszámú algoritmusoknál sem. Ezzel szemben a ma ismert fizikai elveken alapuló gépekre gondoldva teljesen reménytelennek tűnik, hogy 2^{1000} lépést megtehessünk egy emberöltő alatt.

Azt is hangsúlyoznunk kell, hogy nem minden polinom idejű módszer ad igazán hatékony megoldást. Egy cn^k futási idejű algoritmus is lehet elfogadhatatlan gyakorlati szempontból, ha c vagy k nagy. Aligha tekinthetünk jónak mondjuk egy n^{300} lépésszámú módszert. A polinom idejű algoritmus tehát a hatékony módszer *absztrakciójának*, elméletileg kezelhető általánosításának tekintendő. Két fontos tulajdonsága ebből a szempontból, hogy egyrészt tényleg tartalmazza az igazán hatékony (pl. lineáris, kvadratikus idejű) algoritmusokat, másfelől pedig robusztus fogalom az előbb tárgyalt értelemben. Ennek a kis fejtegetésnek az összegzéséül az *elméleti értelemben hatékonyan kezelhető feladatok* a következők:

P:	polinom időben felismerhető nyelvek
FP:	polinom időben számítható függvények

A legtöbb algoritmikus probléma, amivel korábban az adatszerkezetek elemeinél és a gráfalgoritmusok kapcsán találkoztunk, P-beli nyelv felismerésére, vagy általánosabban FP-beli függvény kiszámítására vezet. Például az $x_1, x_2, \ldots, x_n \in I^*$ szavak lexikografikusan rendezett sorrendjének előállítása megfogalmazható mint a következő (nyilvánvalóan FP-beli) f függvény kiszámításának a feladata:

$f : x_1 * x_2 * \ldots * x_n \to y_1 * y_2 * \ldots * y_n$, ahol $y_1 \leq y_2 \leq \ldots \leq y_n$ az x_i szavak lexikografikusan rendezett sorozata. Néhány további ilyen feladat:

1. Kupacépítés
2. Minimális költségű feszítőfa keresése gráfban (Prim és Kruskal módszere)
3. Maximális párosítás keresése páros gráfban (magyar módszer)
4. Maximális folyam számítása egész kapacitásokkal rendelkező hálózatban (Dinic algoritmusa)
5. Gráf csúcsainak 2 színnel való kiszínezése.

Nézzük meg közelebbről mondjuk a magyar módszert! Mint már láttuk, ennek az uniform költsége $O(ne)$, ahol n az éllistával adott G bemeneti gráf pontjainak,

e pedig az éleinek száma. Az éllistának $n + 2e$ cellája van. Az éllista egy cellája kb. $3 \log n$ biten elfér: ebből $\log n$ bit a cellához tartozó csúcs sorszáma. A cellában levő mutatóra mintegy $2 \log n$ bitet szánunk. Ezek elegendően hosszúak lesznek, mivel összesen legfeljebb n^2 cella van a szerkezetben. Az input hossza tehát körülbelül $3(n + 2e) \log n$ bit.

Feladat: Mutassuk meg, hogy a magyar módszer megvalósítható $O(ne \log n)$ logaritmikus költségű RAM-programmal. (Arra kell csak ügyelni, hogy a magyar módszer *egy* elemi lépésének a logaritmikus költsége ne legyen több, mint $O(\log n)$.)

A feladat szerint a magyar módszer logaritmikus költsége az input hosszának négyzetével arányos korlát alatt marad. A módszer tehát tényleg polinom idejű.

Az előzőekkel szemben ellenpontként álljon itt néhány (mai ismereteink szerint) polinom időben nem megoldható feladat. Ezek látszólag egymástól távol eső problémák, mégis algoritmikus szempontból sok közük van egymáshoz. Egyebek között ennek a kapcsolatnak fogunk utánajárni a következőkben.

1. Hamilton-körrel rendelkező gráfok felismerése
2. Utazó ügynök probléma (minimális költségű Hamilton-kör találása)
3. A 3 színnel színezhető gráfok felismerése
4. Maximális méretű klikk keresése gráfokban.

8.3. Nemdeterminisztikus Turing-gépek; az NP nyelvosztály

Az eddig tárgyalt gépmodellek (TG, RAM) valósághoz közelinek mondhatók abban az értelemben, hogy elég hűen tükrözik a számítások tényleges időigényét. A most terítékre kerülő modell ebből a szempontból alaposan elrugaszkodott a valóságtól. Hogy miért foglalkozunk vele mégis? Mert segítségével igen természetesen körülhatárolható egy érdekes és gazdag nyelvosztály. A modellt tehát mint definíciós eszközt érdemes szemlélni.

A *nemdeterminisztikus Turing-gép* (röviden NTG) definíciója abban tér el az egyszalagos Turing-gépétől, hogy a δ átmenetfüggvény nem egyértékű. A δ a lehetséges lépések olykor egynél több elemű véges halmazát jelöli ki:

$$\delta(q, a) \subseteq Q \times T \times \{\text{jobb}, \text{bal}, \text{helyben}\}.$$

Ennek megfelelően a gép esetleg több lehetőségből választhat. A q állapotban az a szalagjel mellett a következő lépést a $\delta(q, a)$ halmazból kell választani.

A gép futása (egy számítási út):

Kiindulási helyzet:
Mint az egyszalagos TG-nél, az $x \in I^*$ input szó a szalagon van, balra igazítva, utána üresjelek. A gép a q_0 kezdőállapotban van, a fej pedig a szalag első celláján.
Egy lépés:
A gép egy pillanatnyi állapotát a belső állapot, a szalag tartalma, valamint a fej helyzete határozza meg. A következő pillanatnyi állapotba úgy jut el, hogy a belső állapotból, valamint a fej alatt található szalagjelből álló párra alkalmazva a δ halmazértékű függvényt, választja annak egy tetszőleges elemét, majd az annak megfelelő belső állapotot veszi fel, miközben a megfelelő szalagjelet írja a fej alatti cellába, majd a megfelelő fejmozgatást végzi el. Ha az adott párra δ nincs értelmezve, akkor a gép megáll.

Példa: Tegyük fel, hogy a gép a q állapotban van, a fej alatti szalagjel a, és $\delta(q, a) = \{(q, a, helyben),\ (q_1, b, jobbra),\ (q_2, c, balra)\}$. Ekkor a gép a három lehetséges lépés közül választhat, mindegyik választás megengedett.

Definíció (input elfogadása): *Az M NTG* elfogadja *az $x \in I^*$ inputot, ha az M-et x bemenettel a kiinduló helyzetből indítva van legalább egy elfogadó (egy elfogadó állapotban véget érő) számítási út.*

A korábbiakkal összhangban jelölje L_M az M által ilyen értelemben elfogadott nyelvet. A definícióból közvetlenül adódik a következő:

Állítás: *Az $x \in I^*$ input szó pontosan akkor nincs L_M-ben, ha az M gépet x inputtal indítva nincs elfogadó számítási út.* □

Az NTG-k számításait néha hasznos egy gyökeres irányított faként elképzelni. A fa csúcsait a gép pillanatnyi helyzeteivel címkézhetjük. Egy csúcsnak annyi fia van, ahány lépés megengedett a csúcsnak megfelelő pillanatnyi helyzetből. A példabeli gép esetén, ha a pillanatnyi helyzetben a belső állapot q, a fej alatti szalagjel a, akkor a csúcsnak három fia van. A fa gyökerét az $x \in I^*$ inputnak megfelelő kiinduló helyzettel címkézzük. A gép pontosan akkor fogadja el az x inputot, ha a fában van olyan gyökértől levélig menő út, melynél a levélhez elfogadó állapot tartozik.

A közönséges Turing-gépek mintájára beszélhetünk időkorlátos nemdeterminisztikus gépekről. Legyen $t : \mathbb{Z}^+ \to \mathbb{Z}^+$ egy függvény, melyre minden $n \in \mathbb{Z}^+$ esetén teljesül a $t(n) \geq n$ egyenlőtlenség.

Definíció ($t(n)$ időkorlátos NTG):
Egy M nemdeterminisztikus Turing-gép $t(n)$ időkorlátos, ha n hosszúságú inputokon M minden számítási út mentén legfeljebb $t(n)$ lépést téve megáll.

Úgy is fogalmazhatunk, hogy egy tetszőleges n hosszúságú $x \in I^*$ input esetén a számításhoz rendelt fa magassága legfeljebb $t(n) + 1$. A követelmény egyaránt korlátozza az elfogadó és az elutasító számítási utak hosszát. Egy időkorlátos NTG tehát egyetlen számítási úton sem kerülhet végtelen ciklusba. A fejezet bevezetőjében említettük, hogy az NTG nem egy valósághű számítási modell. A működésükben éppen ez a költségszámítás jelenti az igazán irreális tényezőt. Nem ismert (és a mai tudásunk szerint nem látszik lehetségesnek) olyan tényleges fizikai megvalósításuk, mely egy $t(n)$ időkorlátos NTG működését $t(n)$-nel arányos időben szimulálni tudná. (Olykor-olykor felröppenek sajtókacsák, állítva, hogy sikerült igazi nemdeterminisztikus gépet konstruálni...)

Az időkorlátos gép fogalmával a kezünkben a $TIME$ osztályok mintájára definiálhatjuk a nemdeterminisztikus időkorlátokkal kijelölt nyelvosztályokat.

Definíció:
$NTIME(t(n)) := \{$*az* $O(t(n))$ *időkorlátos NTG-k által elfogadott nyelvek*$\}$.

NTG-k segítségével fogalmazható meg kényelmesen a számításelmélet egyik legérdekesebb nyelvosztályának, az NP -nek a definíciója.

Definíció: NP $:= \cup_{k\geq 1} NTIME(n^k)$.

Az NP nyelvosztály ugyanúgy épül fel az $NTIME(n^k)$ alakú nyelvhalmazokból, mint a P a $TIME(n^k)$ alakúakból. A definíciók közötti nyilvánvaló formai hasonlóság alapján az NP osztályt a P nemdeterminisztikus megfelelőjének tekinthetjük. Az NP a nemdeterminisztikus polinom idejű Turing-gépekkel felismerhető nyelvekből áll. Az egyszalagos (közönséges) TG-k felfoghatók NTG-nek is, sőt egy $t(n)$ időkorlátos TG tekinthető $t(n)$ időkorlátos NTG-nek is. Így azonnal adódik, hogy $TIME(n^k) \subseteq NTIME(n^k)$, amiből az egyesítésekre térve:

Állítás: P $\subseteq$ NP.□

Az NP nyelvosztály tehát tartalmazza P-t, a hatékonyan kezelhető nyelvek osztályát. A két osztály, a P és az NP közötti analógia korántsem teljes. Nem ismert például, hogy az NP osztály megegyezik-e a coNP nyelvosztállyal (emlékeztetőül: coNP $= \{L \subseteq I^*; I^* \setminus L \in$ NP$\}$). A témakörben dolgozó kutatók azt sejtik, hogy a két osztály különböző. Szintén nyitott kérdés, hogy P megegyezik-e NP-vel. Itt is a nemleges válasz látszik valószínűnek. A (P = NP?) kérdés **a számításelmélet egyik legfontosabb megoldatlan problémája**.

Állítás: P $\subseteq$ NP $\cap$ *co*NP.

Bizonyítás: P $\subseteq$ NP miatt coP $\subseteq$ coNP. Ezután az állítás azonnal adódik a P $=$ coP egyenlőségből. □

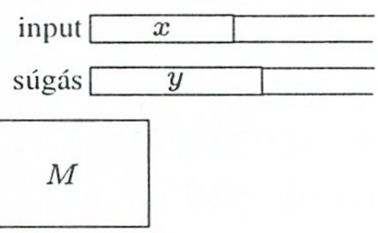
Egyszerű alkalmazásként megjegyezzük, hogy a P $=$ NP egyenlőségből NP $=$ coNP következne.

A nemdeterminisztikus számítások egy determinisztikus modellje

Először megsejtesz valamit. Ne nevessetek, ez a legfontosabb lépés. Utána kiszámítod a következményeket... RICHARD P. FEYNMAN
a fizikai felfedezésekről

Célunk itt nyelvek nemdeterminisztikus felismerésének determinisztikus modelljét adni. Ezzel végső soron kényelmes eszközt kapunk nyelvek (feladatok) NP-be tartozásának igazolására, amely jól használható konkrét nyelvek esetén.

Legyen M kétszalagos (közönséges) TG. Tegyük fel, hogy M inputja két részből áll, egyik része – mondjuk $x \in I^*$ – az első szalagon van, a másik része $y \in I^*$ a másikon. Az utóbbi, az y-t tartalmazó szalag csak olvasható. Ezt nevezzük az M *súgásszalagjának*.



A korábbi definícióval összhangban az M által felismert L_1 nyelv azon (x, y) szópárok halmaza ($x, y \in I^*$), melyekkel a kezdő helyzetből elindítva M elfogadó állapotban áll meg.

Definíció (nemdeterminisztikus felismerés): *Az M által nemdeterminisztikusan felismert L nyelv a következő:*

$$x \in L \text{ akkor, és csak akkor, ha van olyan } y \text{ súgás, hogy } (x, y) \in L_1.$$

Az M gép által nemdeterminisztikusan felismert L nyelvbe tehát pontosan azok az $x \in I^*$ szavak tartoznak, melyek kiegészíthetők alkalmas y-nal (súgással) úgy, hogy az (x, y) párt az M elfogadja. Itt fontos mozzanat, hogy a jó súgásnak csak a *létezése* követelmény. Semmit sem teszünk fel arról, hogy adott x-hez

miként található alkalmas y. Az y szót szokás még az $x \in L$ állítás *tanújának*, vagy az x elfogadásához vezető *sejtésnek* is nevezni. A *súgás* és *sejtés* szavakból kiszüremlő bizonytalanság arra hivatott utalni, hogy az y-nal szemben nincs kiszámíthatósági követelmény.

A következő tétel szerint az NP-beli L nyelveknél az M egy polinom idejű TG és a tanú (súgás) hossza is polinomkorlátosnak vehető.

Tétel (tanú-tétel): *Egy $L \subseteq I^*$ nyelvre a következő két állítás egyenértékű:*
(a) $L \in$ NP.
(b) Van olyan $c > 0$ állandó, továbbá egy $L_1, \in$ P nyelv, mely olyan $(x, y) \in (I^*)^2$ párokból áll, hogy $|y| \leq |x|^c$ és $x \in I^*$ esetén $x \in L$ pontosan akkor, ha van $y \in I^*$ úgy, hogy $(x, y) \in L_1$.

Bizonyítás: (Vázlat) $(a) \Rightarrow (b)$: Az $L \in$ NP feltétel miatt létezik egy n^{c_1} időkorlátos N NTG, mely felismeri L-et. Tegyük fel, hogy N-nek egy lépésnél legfeljebb d elágazási lehetősége van, vagyis d a $\delta(q, a)$ halmazok maximális elemszáma. Egy $c > 0$ szám és egy olyan M polinomkorlátos TG létezését kell megmutatnunk, mely (x, y) alakú inputokkal dolgozik, és $x \in L$ pontosan akkor teljesül, ha van olyan $y \in I^*$, $|y| \leq |x|^c$, hogy $(x, y) \in L_1 = L_M$.

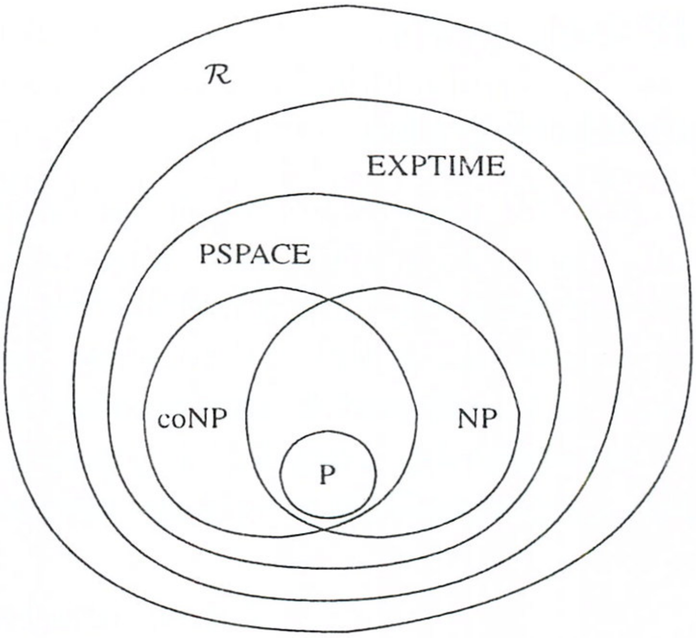
Legyen $x \in L$, $|x| = n$. Erre az inputra egy olyan $y = y_1 y_2 \cdots y_m$ alakú sorozatot ($m \leq n^{c_1}$) fogunk jó súgásnak tekinteni, mely az N egy elfogadó számítási útját írja le az x input mellett. Az y_j (bináris) szavak 1 és d közé eső egészek kódjai, amiből $|y| \leq n^{c_1} \lceil \log_2(d + 1) \rceil \leq n^c$ alkalmas c konstanssal. Az y_j által ábrázolt szám mondja meg, hogy a j-edik lépésben N melyik lehetséges lépését kell választani (az adott helyzetben szóba jövő legfeljebb d lehetséges lépés közül) az elfogadó számítási úton. Az M TG egy „nagy" lépése a következő: a súgásszalagról megnézi, hogy N melyik lépését kell megtenni (ez legfeljebb $\lceil \log_2(d + 1) \rceil$ bitet jelent), majd ezt meglépi. Ez N-től függő konstans időben megtehető. Az M pontosan akkor álljon meg, ha N szimulált „lépése" megállás, fogadja el az inputot, ha ez a megállás N elfogadó állapotában történt. Az M álljon meg elutasító állapotban akkor is, ha a súgás következő darabja nem értelmezhető N legális lépéseként. A fentiek alapján M az (x, y) összetett input hosszában lineáris ideig működik. Világos az is, hogy $x \in L$ esetén van olyan jó y súgás, melyre $|y| \leq n^c$. Ha viszont $x \notin L$, akkor $(x, y) \notin L_1 = L_M$ tetszőleges $y \in I^*$-ra. Ekkor ugyanis az (x, y) párt M vagy azért utasítja el, mert y nem kódja N legális számításának, vagy pedig azért, mert y elutasító számítási út kódja; elfogadóé nem lehet, hiszen elfogadó út nem létezik.
$(b) \Rightarrow (a)$: Egy $x \in L$ inputhoz a feltétel szerint van rövid tanú (súgás): ha $|x| = n$, akkor van olyan legfeljebb n^c hosszúságú y, hogy $(x, y) \in L_1$. Az ilyen rövid jó súgások száma legfeljebb $\leq 2^{n^c}$. Megmutatható, hogy ennyi lehetőség

nemdeterminisztikusan n^c lépésben végignézhető. Ezt itt nem részletezzük (de a következő feladatban körvonalazzuk). □

Feladat: Tegyük fel, hogy az M egy n^c időkorlátos TG, mely a tételbeli L_1 nyelvet ismeri fel. Ebből készítünk egy N NTG-t a következők szerint. Az N állapotai egyezzenek meg az M állapotaival, elfogadó (elutasító) állapotai az M elfogadó (elutasító) állapotaival. Az N-nek legfeljebb két lehetséges lépése lesz a q állapotban és az a szalagjel olvasásakor. Ezek legyenek azok a lépések, melyeket M tesz abban a két esetben, amikor
M állapota q, az inputfej alatti jel a, a súgásfej alatti jel 0;
M állapota q, az inputfej alatti jel a, a súgásfej alatti jel 1.
Természetesen N-nek csak egy szalagja van, így az M lépéseiből csak az állapotváltoztatást és az input szalaggal kapcsolatos teendőket az állapotváltoztatást és az input szalaggal kapcsolatos teendőket kell megtenni. Bizonyítandó, hogy N éppen L-et ismeri fel, és n^c időkorlátos.

Megjegyzések: 1. A (b) állításban az $|y| \leq |x|^c$ feltétel helyettesíthető az $|y| = |x|^c$ feltétellel. Ugyanis egyrészt feltehetjük, hogy c egész; ha kell, helyettesíthetjük $\lceil c \rceil$-vel. Másrészt egy y tanút kiegészíthetünk a végére ragasztott tetszőleges bitekkel, hogy a hossza éppen $|x|^c$ legyen.
2. A tanú-tételből következik, hogy NP $\subseteq$ PSPACE. Legyen ugyanis $L \in$ NP, x egy input szó, $|x| = n$. A legfeljebb n^c hosszú $y \in I^*$ szavakat mint tanújelölteket sorban generáljuk, és minden (x, y) szópárra lefuttatjuk M-et, egészen addig, amíg vagy elfogadó számítást kapunk (ekkor $x \in L$), vagy elfogynak a lehetséges jelöltek (ekkor $x \notin L$). Ez minden egyes y szóra $O(n^c)$ időt jelent. Ha egy y szóval végeztünk, akkor a kapott részeredményeket eldobhatjuk, és kezdhetjük az M számítását elölről a következő tanújelölttel. A szükséges adminisztráció megoldható egy n^c bites számláló segítségével. Az n^c hosszú tanújelölteket tekinthetjük úgy, mint n^c bites (binárisan ábrázolt) számokat. A tanújelöltek szisztematikus végignézése azt jelenti, hogy nullától $2^{n^c} - 1$-ig végiglépkedünk az egészeken, mindig az eggyel nagyobbat véve. Abban a kellemes – és igen egyszerű – helyzetben vagyunk, hogy ehhez a lépkedéshez nem kell sok memória. A következő szám (tanújelölt) könnyen és gyorsan megkapható pusztán az előző ismeretében.

Az eddig megismert bonyolultsági osztályok tartalmazási viszonyait szemlélteti a következő vázlat.



Nem tudjuk, hogy a P osztály valódi része-e a PSPACE osztálynak. A sejtés a korábbiakkal összhangban az, hogy igen. Az eddig vett nyitott kérdések közül ez a leggyengébb, amint azt az alábbi egyszerű következtetési lánc mutatja:

$$\mathrm{NP} \neq \mathrm{coNP} \Rightarrow \mathrm{P} \neq \mathrm{NP} \Rightarrow \mathrm{PSPACE} \neq \mathrm{P}$$

Egy „kép" az NP-beli nyelvekről:

A tanú-tétel determinisztikus jellemzése az egyik leghasznosabb eszköz nyelvek NP-be tartozásának belátására. Gondolva a polinom idejű számítások robusztusságára is, az L_1 nyelvről (döntési problémáról) elég látni, hogy kezelhető polinom idejű „programmal". Nem kell a nehézkes TG-modellel dolgozni.

Az NP nyelvosztály determinisztikus leírásának elemeit segíthet megjegyezni az alábbi táblázatban vázolt helyzet.

Bíró:	M polinom idejű algoritmus (pl. TG, RAM program)
Vádlott:	$x \in I^*$
Tanú:	$y \in I^*$, $\lvert y\rvert \leq \lvert x\rvert^c$
Vád:	Az állítás, miszerint $x \in L$.

Az M algoritmus (a kissé türelmetlen bíró, akinek időkorlátja van) nem tesz mást, mint ellenőrzi, hogy az y tanú(vallomás) tényleg bizonyítja-e a vádat, ami az $x \in L$ állítás. A tétel L_1 nyelve azon *(vádlott,tanú)* párokból áll, amelyeknél a bíró szerint a *tanú* bizonyítja a *vádlott*ról a vádat. Nem kell foglalkoznia a bizonyíték előtalálásával. Ez – a mai tudásunk szerint – sok érdekes esetben nem is tehető meg polinom időben. Csak az eléje tárt bizonyíték helyességével kell törődnie. Ezt a szemléletet fogjuk használni a következő példáknál.

8.4. Néhány NP-beli nyelv

8.4.1. 3 színnel színezhető gráfok

Rögzítsük gráfok egy nem túl pazarló kódolását bitsorozatokkal. Például egy n szögpontú (irányított, vagy irányítatlan) gráf adjacencia-mátrixát tárolhatjuk egy n^2 hosszúságú bitsorozatként. Egy ilyen kódolás eredményeként a gráfokat $\{0,1\}^*$-beli szavak reprezentálják, gráfok halmazainak pedig nyelvek felelnek meg. Emlékeztetőül: a $G = (V, E)$ gráf k színnel színezhető, ha a csúcsaihoz i egészeket (színeket) rendelhetünk úgy, hogy $1 \leq i \leq k$, és ha $(v, w) \in E$, akkor v és w különböző színűek. Legyen 3-SZÍN a 3 színnel színezhető gráfok kódjaiból álló nyelv.

Állítás: *3-SZÍN* $\in$ NP.

Bizonyítás: Elegendő megmutatni, hogy a három színnel való színezhetőségnek van rövid és hatékonyan ellenőrizhető tanúja. Egy n szögpontú 3 színnel színezhető G gráf jó színezése alkalmas tanú lesz. Egy ilyen színezés leírható $2n$ bittel (például legyen 01=piros, 10=sárga, 11=zöld). A bíró ellenőrzi, hogy az adott színezés jó-e. Ez a feladat polinom időben megoldható.

A tanú-tétel alkalmazásakor a G gráf felel meg az x inputnak, a színezés az y tanú. Az L_1 nyelv $(G, \textit{színezés})$ alakú párokból áll, ahol a *színezés* egy helyes 3-színezése G-nek. A bírót jelentő M algoritmusnak csak annyit kell tudnia, hogy ellenőrzi a két komponens harmóniáját: hogy a javasolt színezés tényleg jó-e.

Figyeljük meg, hogy tényleg teljesülnek a tanú-tétel (b) részének a követelményei. Ha $G \in$ 3-SZÍN, akkor van $(G, \textit{színezés})$ alakú pár L_1-ben. Ha $G \notin$ 3-SZÍN, akkor pedig nem létezhet ilyen pár. □

Feladat: Adjunk $O(n^2 \log n)$ logaritmikus költségű RAM programot, mely megoldja a bíró feladatát: adott n szögpontú G gráfra és a csúcsok egy adott színezésére eldönti, hogy utóbbi 3 színezése-e G-nek.

8.4.2. Hamilton-körrel rendelkező gráfok

A G irányítatlan gráf egy köre *Hamilton-kör*, ha abban G minden csúcsa pontosan egyszer szerepel. Legyen H a Hamilton-kört tartalmazó gráfokból álló nyelv.

Állítás: $H \in$ NP.

Bizonyítás: A $G \in$ H állításnak rövid tanúja egy Hamilton-kör. A Hamilton-kör leírható a csúcsoknak a kör mentén való bejárási sorrendjével, ami $O(n \log n)$

bitet jelent, ha G-nek n csúcsa van. A bíró nyelve (L_1 a tanú-tételben) $(G; [v_1, v_2, \ldots, v_n])$ alakú párokból áll, ahol a $[v_1, v_2, \ldots, v_n]$ Hamilton-köre a G gráfnak. A bíró dolga egyszerű: ellenőrzi, hogy tényleg a G köre-e a tanú, és hogy G minden csúcsa pontosan egyszer szerepel-e a listán. Ezek a feladatok polinom időben megoldhatók. □

Feladat: Adjunk minél hatékonyabb RAM programot a bíró feladatának megoldására.

A G irányított gráf egy irányított köre *Hamilton-kör*, ha abban G minden csúcsa pontosan egyszer szerepel. Legyen IH az irányított Hamilton-kört tartalmazó irányított gráfok nyelve. Az előzőekhez hasonlóan látható, hogy IH $\in$ NP.

8.4.3. Síkba rajzolható gráfok

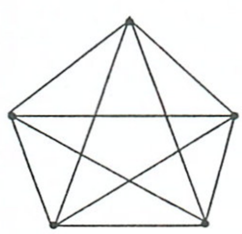
Egy gráf *síkba rajzolható*, ha a pontjainak kölcsönösen egyértelműen megfeleltethetők síkbeli pontok, az éleknek pedig a megfelelő csúcsokat összekötő egyenesszakaszok úgy, hogy a különböző szakaszok legfeljebb csak a végpontjaikban találkozhatnak. Legyen *Sík* a síkba rajzolható gráfok nyelve. Mint látni fogjuk, ennek a nyelvnek a komplementere is NP-beli. Az ilyen nyelvek esetén annak a ténynek is van rövid és hatékonyan ellenőrizhető tanúja, hogy egy input szó *nem* tartozik a nyelvbe.

Definíció (jól karakterizált nyelv): *Az $L \subseteq I^*$ nyelv jól karakterizált, ha $L \in$* NP $\cap$ *co*NP.

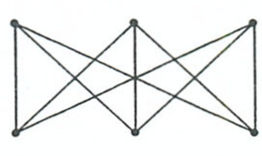
Állítás: *A Sík nyelv jól karakterizált.*

Bizonyítás: Egy gráf síkba rajzolhatóságának a tanúja egy síkba rajzolás. (Megmutatható, hogy ha $G \in$ *Sík*, akkor van a G-nek olyan síkba rajzolása is, ahol a pontok koordinátái $|V(G)|$-nél nem nagyobb természetes számok). A síkba rajzolás helyességét könnyű ellenőrizni. Lényegében végpontjaikkal adott szakaszok metszéseit kell számolni. Ezzel beláttuk, hogy *Sík* $\in$ NP.

A *Sík* $\in$ coNP állítás igazolásához azt kell megmutatni, hogy a $G \notin$ *Sík* ténynek is van hatékonyan ellenőrizhető tanúja. Ez pedig következik K. Kuratowski nevezetes tételéből : a G gráf nem síkba rajzolható $\Leftrightarrow$ G topologikusan tartalmaz teljes ötszöget vagy 3 ház – 3 kút gráfot.



teljes ötszög



3 ház – 3 kút

A $G \notin$ *Sík* ténynek egy G-beli topologikus teljes 5-ös vagy 3 ház – 3 kút gráf lehet a tanúja. A tanú tehát a tiltott részgráfot kijelölő 5 vagy 6 pontból áll és az ezek közül a megfelelőeket összekötő, közös belső pontot nem tartalmazó G-beli utakból. A bírónak csupán a megfelelő pontpárok közötti utak meglétét kell ellenőriznie, és még azt, hogy az utak diszjunktak. Ezek egyszerűen és gyorsan verifikálhatók. □

Megjegyzések: 1. Valójában a Kuratowski-tételből az erősebb *Sík* $\in$ P állítás is következik. A síkgráfok tehát polinom időben felismerhetők. Ezen felül a síkba rajzolásra is van hatékony algoritmus. Ezeket az állításokat nem részletezzük.
2. Szigorúan véve a *Sík* nyelv komplementere nem pontosan a síkba nem rajzolható gráfokból áll. A komplementer nyelvbe tartoznak még azok a szavak is, melyek nem kódjai gráfoknak. Ezekről azonban feltehetjük, hogy könnyen felismerhetők. Arról van csupán szó, hogy értelmes kódolást választottunk, aminél nem okoz gondot a hibás kódok felismerése. Ezt figyelembe véve a *Sík* $\in$ coNP állítás érdemi része tényleg az, hogy a síkba nem rajzolhatóságnak van hatékony tanúja.

Itt elgondolkodhatunk egy kicsit azon, hogy a 3-SZÍN és a H nyelvek hogyan viszonyulnak a coNP osztályhoz. Van-e annak rövid és hatékonyan ellenőrizhető tanúja, hogy egy gráf *nem* színezhető 3 színnel, illetve, hogy *nem* tartalmaz Hamilton-kört? Ilyen tanúk nem ismeretesek, és a szakértők úgy sejtik, hogy nem is léteznek. Ha igazuk van, akkor ezek a nyelvek mutatják az NP és a coNP osztályok különbözőségét.

Itt ismét visszautalunk Merlin szerencséjére az első fejezetből: a házasítási feladat megoldhatatlanságának volt rövid, a király által is gyorsan átlátható bizonyítéka (egy király legfeljebb polinom időt hajlandó ilyesmire pazarolni). Egy König-akadály a megoldhatatlanság gyors bizonyítéka. A varázsló nem lenne ilyen könnyű helyzetben, ha arról kellene Arthurt meggyőznie, hogy egy nagy gráfban *nincs* Hamilton-kör. Az NP-beli feladatok éppen azok, ahol az igenlő válasz esetén a varázsló elkerülheti a türelmetlen király haragját.

A gráfalgoritmusokról szóló fejezetben találkoztunk néhány minimax tétellel (Ford–Fulkerson, Menger, König). Az ilyen eredmények gyakran hordozzák azt az algoritmikus jelentést, hogy a megfelelő nyelv jól karakterizált. Példaként nézzük a Ford–Fulkerson-tételt; a másik kettő meggondolását az olvasóra hagyjuk.

Feladat: Álljon a $Folyam$ nyelv azon $(\mathcal{H}, k)$ alakú párokból, amelyekre $\mathcal{H} = (G, s, t, c)$ egy egész kapacitásokkal rendelkező hálózat, $k \in \mathbb{Z}^+$, és $\mathcal{H}$-ban van olyan f folyam, aminek az értéke legalább k. Mutassuk meg, hogy $Folyam \in$ NP $\cap$ coNP. (Legyen $x \in I^*$. Az $x \in Folyam$ tény tanúja egy $\mathcal{H}$-beli f folyam, melyre $|f| \geq k$; $x \notin Folyam$ tanúja pedig egy $\mathcal{H}$-beli vágás, aminek a kapacitása kisebb, mint k.)

Megjegyezzük, hogy a feladatban foglaltnál erősebb $Folyam \in$ P állítás is igaz; Dinic módszerével polinom időben találhatunk egy $\mathcal{H}$-beli f^* maximális folyamot. Nyilvánvalóan $(\mathcal{H}, k) \in Folyam$ egyenértékű azzal, hogy $|f^*| \geq k$.

8.4.4. A prímszámok nyelve

A $p > 1$ egész szám *prímszám*, ha a pozitív egészek közül csak az 1 és p számokkal osztható maradék nélkül. A $p > 1$ egész szám *összetett szám*, ha nem prímszám.

Jelölje Π a (binárisan ábrázolt) prímszámok nyelvét. Itt a komplementer tulajdonságnak, az összetettségnek van egyszerű tanúja. Ha m összetett szám, akkor ezt egy k valódi osztója ($1 < k < m$) tanúsítja. A k ismeretében m összetettsége az $m : k$ osztás elvégzésével igazolható. Tegyük fel, hogy m egy n-bites szám (az input hossza n). Az elemi iskolából ismert osztó algoritmussal az $m : k$ osztás $O(n^2)$ bit-művelettel elvégezhető. Van tehát polinom idejű bíró. Ezzel beláttuk, hogy $\Pi \in$ coNP. Lényegesen bonyolultabb a következő állítás.

Tétel (V. R. Pratt, 1975): $\Pi \in$ NP *(tehát Π jól karakterizált).*

Pratt tételének bizonyításához szükségünk lesz egy számelméleti lemmára. Emlékeztetőül: $a \equiv b \pmod{m}$ jelöli azt a tényt, hogy $b - a$ osztható m-mel (a, b, m egész számok).

Lemma: *Legyen $p \geq 2$ egy egész szám. A p pontosan akkor prímszám, ha van olyan $1 \leq g < p$ egész, melyre teljesülnek az alábbiak:*

1. $g^{p-1} \equiv 1 \pmod{p}$,

2. $g^{\frac{p-1}{r}} \not\equiv 1 \pmod{p}$ *minden r prímszámra, melyre* $r | p-1$.

□

A lemma bizonyítását mellőzzük. A számelmélet elemeiben jártas olvasó számára megjegyezzük, hogy g a modulo p maradékosztályok multiplikatív csoportjának egy generátoreleme (ún. primitív gyök).

Szükségünk lesz még a *gyors hatványozás* algoritmusára. Ez egy ősrégi ötleten[1] alapul, és fontos építőköve a számítógépes aritmetikának. Gyors hatványozással egy a^m alakú hatvány (m egy pozitív egész) legfeljebb $2\log_2 m$ szorzással kiszámítható. Írjuk fel ugyanis az m kitevőt kettes számrendszerben $m =$

[1] Az i.e. 1650 táján íródott egyiptomi *Rhind-papirusz* szorzás helyett összeadással, vagyis a hatványozás helyett a szorzás elvégzésére ismerteti a módszert.

$e_0 + e_1 2^1 + e_2 2^2 + \cdots + e_k 2^k$, $k \leq \log_2 m$ és $e_j \in \{0,1\}$. Számítsuk ki ismételt négyzetre emelésekkel sorra az a^{2^j} hatványokat $j = 1, 2, \ldots, k$. Ez k szorzást jelent. Végül szorozzuk össze az a^{2^j} hatványokat azokra a j értékekre, melyekre $e_j = 1$. Ez legfeljebb k további szorzást igényel. Úgy fogalmazhatunk, hogy a szorzások száma polinomiális (sőt lineáris) az m kitevő *méretében*. Ha a egész szám, akkor az a^m végeredmény mérete exponenciális is lehet a méretéhez képest. Nem ilyen rossz a helyzet, ha a gyors hatványozást $a^m (mod\, m_1)$ alakú maradékosztályok kiszámítására használjuk. Ezt a feladatot *moduláris hatványozásnak* nevezik. Ekkor minden egyes szorzás után redukálhatjuk a szorzatot modulo m_1. Ha itt a és m_1 legfeljebb n-bites számok, akkor a számítás során fellépő egészek hossza legfeljebb $2n$ bit lesz. A gyors hatványozás tehát polinom idejű algoritmust ad a moduláris hatványozás feladatára (binárisan ábrázolt a, m, m_1 bemenet esetén).

Pratt tételének bizonyítása: Olyan tanút kell javasolnunk, amelynek birtokában hatékonyan igazolható, hogy az inputként adott p prímszám. Próbáljunk a lemma által sugallt úton járni: adott g szám, valamint a $p-1$ egész $r_1, \ldots, r_k$ prímosztói ismeretében *gyors hatványozással* ellenőrizhetők a lemma kongruencia-feltételei. Első közelítésben a g és az $r_1, \ldots, r_k$ számokat tekintjük tanúnak. Tanúsítanunk kell még, hogy $r_1, \ldots, r_k$ éppen a $p-1$ prímosztói. Ennek a könnyebbik része ellenőrizni, hogy $p-1$ előáll-e az r_i számok hatványainak szorzataként. Tanúsítani kell ezen kívül, hogy az r_i számok is prímek. Ezekhez ugyanolyan szerkezetű tanút használunk, mint a p-hez. Például az r_1-hez szükségünk lesz egy alkalmas g_1 számra, valamint az $r_1 - 1$ prímosztóira és azok tanúira is. Ez tehát egy rekurzíve megadott szerkezetű tanú lesz. Megmutatható, hogy ha a p input egy n bites szám, akkor az így felépített tanú összmérete $O(n^2)$, és a bíró algoritmusa n-ben polinomiális idejű. □

Megjegyzés: Nyitott kérdés, hogy $\Pi \in$ P teljesül-e, azaz van-e polinom idejű módszer a prímtulajdonság (összetettség) eldöntésére. Ha a válasz nemleges, akkor P $\neq$ NP $\cap$ coNP. A legjobb ismert prímfelismerő módszer (L. M. Adleman, C. Pomerance, R. S. Rumely, 1983) futási ideje n bites input esetén $n^{c \log \log n}$.

8.4.5. A felismerés és a keresés kapcsolata (prímtényezős felbontás)

A nyelvfelismerési problémák olyan feladatoknak felelnek meg, amelyeknél a válasz egyetlen bittel (igen-nem) kifejezhető. Gyakrabban találkozunk olyan problémákkal, amikor a várt eredmény összetett, például amikor egy mennyiség optimumát kell meghatározni, vagy egy halmaz adott tulajdonságú elemeit kell megkeresni. Ezeket az általánosabb feladatokat szokásos (bár nem túl pontos) szóhasználattal *keresési feladatoknak* nevezik. A keresési feladatokat felfoghatjuk valamely

$f : I^* \to I^*$ függvény kiszámításának problémájaként. Gyakran előfordul, hogy f számítása hatékonyan visszavezethető egy felismerési problémára (egy nyelv felismerésére). Erre a jelenségre szeretnénk egy példát mutatni a következőkben.

Álljon az F nyelv azokból az (a, c) pozitív egész párokból, melyekre igaz, hogy a-nak van c-nél nem nagyobb valódi osztója:

$$F = \left\{ (a, c) \;\middle|\; \begin{array}{l} 1 < c \leq a \text{ egészek és van olyan } 1 < b \leq c \text{ egész,} \\ \text{melyre } b \text{ osztója } a\text{-nak} \end{array} \right\}.$$

Állítás: $F \in \text{NP} \cap co\text{NP}$.

Bizonyítás: Az $(a, c) \in F$ ténynek egy jó b érték lesz a tanúja. A bíró ellenőrzi, hogy b osztja-e a-t, és az $1 < b \leq c$ egyenlőtlenségeket. Az $(a, c) \notin F$ ténynek alkalmas tanúját adják az a prímtényezős felbontásában szereplő $p_1, \ldots p_k$ prímek, valamint a p_i számok prímtulajdonságának tanúi (Pratt tétele). A bíró ellenőrzi, hogy a p_i számok prímek-e, a előáll-e ezek hatványainak szorzataként, és végül, hogy teljesülnek-e a $c < p_i$ egyenlőtlenségek. □

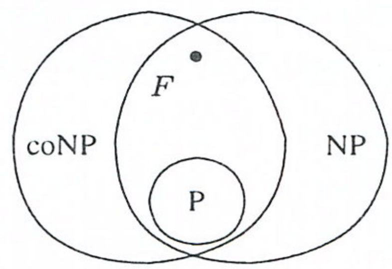
A kapcsolódó keresési feladat a *prímtényezős felbontás* problémája: egy binárisan adott a egésznek keressük a prímosztóit. Ez egyike a legtöbbet vizsgált algoritmikus kérdéseknek. Hatékony algoritmus kidolgozásával már Legendre és Gauss is foglalkoztak – a cél felől tekintve nem sok sikerrel. Ezekből a kutatásokból sok fontos és szép számelméleti eredmény született, de polinom idejű módszert máig sem sikerült találni. Az eddigi leggyorsabb algoritmus D. Shanks *törpe lépés - óriás lépés módszere*; ennek időigénye n bites inputon $c2^{n/4}$. A következő állítás szerint szoros összefüggés van az F és a prímtényezős felbontás algoritmikus nehézsége (bonyolultsága) között. Ha F hatékonyan felismerhető, akkor a prímtényezős felbontás problémája is hatékonyan megoldható.

Tétel: *Ha $F \in$ P igaz lenne, akkor {prímtényezős felbontás} $\in$ FP is igaz lenne.*

Bizonyítás: Meg fogjuk mutatni, hogy ha van gyors (polinom idejű) E eljárásunk F felismerésére, akkor ennek segítségével polinom időben megtalálhatjuk az a input egész prímtényezőit. Először az E eljárás egy hívásával teszteljük, hogy $(a, a-1) \in F$ teljesül-e. Ha nem, akkor megállhatunk, mert a prím, hiszen nincs 1-től és a-tól különböző pozitív osztója. Ellenkező esetben bináris kereséssel meghatározzuk a legkisebb olyan c értéket, melyre $(a, c) \in F$. Ez az E legfeljebb $\log_2 a$ számú hívásával megoldható. Először az $(a, \lceil a/2 \rceil)$ párt teszteljük, ha ez F-beli, akkor megyünk tovább lefelé, stb. A kapott c egész nyilvánvalóan az a legkisebb prímosztója; különben c nem volna minimális. Ezután az $a \leftarrow \frac{a}{c}$ értékkel ismételjük a fenti eljárást.

Az a egy prímosztóját így $O((\log_2 a)^d)$ időben leválasztjuk, ahol $d > 0$ egy állandó, hiszen a feltétel szerint egy E-hívás időigénye polinomiális az input hosszát jelentő $\log a$-ban. Végeredményben az a összes prímosztóját megkapjuk $O((\log a)^{d+1})$ költséggel, mert a prímosztók száma legfeljebb $\log_2 a$. □

Megjegyzés: Valójában nem ismert polinom idejű módszer az F nyelv felismerésére. A szakértők közül sokan úgy vélik, hogy $F \notin$ P. Ebből következne, hogy P $\neq$ NP $\cap$ coNP. Ennek a negatív ténynek, mármint hogy egy feladatra *nincs* hatékony módszer, fontos alkalmazásai vannak a biztonságos (titkos) kommunikáció területén. Vannak olyan kommunikációs protokollok, amelyeknél az üzenetek megfejtésének feladata összehasonlítható az F felismerésének feladatával. Ha a rendszer titkos kulcsait nem ismerő kívülálló meg tudná fejteni az üzeneteket, akkor F-et is kezelni tudná. A következő ábra az F nyelv (vélt) helyét mutatja a bonyolultsági osztályok térképén.



8.5. Karp-redukció, NP-teljesség

Az F nyelv és a prímfelbontás kapcsolatának taglalásakor tulajdonképpen összehasonlítottuk a két feladatot. Az eredmény úgy is fogalmazható, hogy a prímfelbontás problémája - polinom idejű extra munka erejéig - *nem nehezebb*, mint az F nyelv felismerése. Azt is mondhatjuk, hogy a prímfelbontás feladatát *visszavezettük* F felismerésének a feladatára. A kapott felbontó algoritmus futása során többször hívtuk az E eljárást. Most egy ennél szigorúbb visszavezetésfogalmat adunk meg.

Definíció: *Az $f : I^* \to I^*$ leképezés az $L_1 \subseteq I^*$ nyelv* Karp-redukció*ja az $L_2 \subseteq I^*$ nyelvre, ha*

1. Tetszőleges $x \in I^*$ szóra $x \in L_1$ pontosan akkor teljesül, ha $f(x) \in L_2$;

2. $f \in FP$, azaz f polinom időben számítható.

A továbbiakban $L_1 \prec L_2$ jelöli azt a tényt, hogy L_1-nek van Karp-redukciója L_2-re. A definíció első követelménye azt fejezi ki, hogy f hűségesen transzformálja a nyelvbe tartozást. Ha $x \in I^*$ benne van L_1-ben, akkor $f(x)$ benne lesz L_2-ben; és fordítva: ha x nincs L_1-ben, akkor $f(x)$ sincs L_2-ben. A második követelmény szerint az f transzformáció gyorsan számítható. Van olyan csak az f-től függő $c > 0$ állandó, hogy az $f(x)$ szó $|x|^c$ lépésben megkapható x-ből. Ennek folyománya például, hogy $f(x)$ nem lehet túl hosszú: $|f(x)| \leq |x|^c$.

Egy Karp-redukcióval az L_1 felismerésének a problémáját visszavezetjük az L_2 felismerésének a problémájára. A Karp-redukció jelentős segítséget ad, hogy eligazodhassunk az NP-beli nyelvek dzsungelében. Elsősorban annak a megállapítására használható, hogy egy nyelv (döntési feladat) nehéz. A jellemző alkalmazási helyzetben egy eddig ismeretlen L' nyelvet hasonlítunk össze egy már ismert (mondhatni: hírhedt) L nyelvvel. Egy $L \prec L'$ Karp-redukció arra enged következtetni, hogy L' nehéz nyelv, feltéve, hogy ez igaz L-re. Ezt alapozza meg a következő állítás. Látni fogjuk, hogy $L_1 \prec L_2$ esetén L_1 hatékonyan felismerhető, ha van hatékony módszerünk L_2 felismerésére. Az utóbbi eljárást egy inputra csak egyszer kell hívni. Egy $L_1 \prec L_2$ redukció létezéséből arra következtethetünk, hogy L_2 felismerése – polinom idejű extra munka erejéig – legalább olyan nehéz, mint L_1-é.

Első példaként ismertetünk egy Karp-redukciót az irányított Hamilton-kört tartalmazó gráfok nyelvéről az irányítatlan esetre. Egy olyan ötlet lesz segítségünkre, mellyel az irányított élek kódolhatók irányítatlan élekkel.

Állítás: $IH \prec H$.

Bizonyítás: Legyen $G = (V, E)$ egy irányított gráf. Ebből egy $G' = (V', E')$ irányítatlan gráfot készítünk úgy, hogy G ismeretében G' gyorsan megépíthető legyen; továbbá G-ben pontosan akkor legyen irányított Hamilton-kör, ha G'-ben van irányítatlan Hamilton-kör.

Evégből vegyünk G minden $v \in V$ csúcsához egy 2 hosszúságú irányítatlan utat: $v_{be} - v_* - v_{ki}$. Az $u \to v \in E$ élnek feleltessük meg az $u_{ki} - v_{be}$ élet. Formálisan:

$$
\begin{aligned}
V' &= \{v_{be}, v_*, v_{ki} \mid v \in V\}, \\
E' &= \{(v_{be}, v_*), (v_*, v_{ki}) \mid v \in V\} \cup \{(u_{ki}, v_{be}) \mid u \to v \in E\}.
\end{aligned}
$$

Ha G-nek n csúcsa és e éle van, akkor G'-nek $3n$ csúcsa és $2n+e$ éle lesz. Világos, hogy G ismeretében G' polinom időben, azaz $(n + e)^c$ lépésben megkapható. A Karp-redukció második (hatékonysági) követelménye tehát teljesül.

Nézzük most az első feltételt. G minden F irányított Hamilton-körének természetesen megfeleltethető G' egy F' Hamilton-köre. Az F egy $u \rightarrow v$ élének az F'-ben az $u_* - u_{ki} - v_{be} - v_*$ út felel meg. Ha tehát $G \in$ IH, akkor $G' \in$ H. A fordított irányú követelményhez tegyük fel, hogy G'-ben van egy $F' \subseteq E'$ Hamilton-kör. Járjuk be ezt a kört egy csillagos pontjából kezdve úgy, hogy a *ki* indexű szomszéd felé lépjünk először. Ekkor a körön a bejárás szerint szomszédos két csillagos pont közötti útdarab csak $u_* - u_{ki} - v_{be} - v_*$ alakú lehet. Ebből pedig azonnal adódik, hogy az

$$F = \{u \rightarrow v \mid \{u_{ki}, v_{be}\} \in F'\}$$

élhalmaz irányított Hamilton-köre G-nek. A $G \mapsto G'$ megfeleltetés tényleg Karp-redukció. □

A $G \mapsto G'$ leképezést szemlélve elmondhatjuk, hogy IH felismerése nem nehezebb, mint a H nyelvé. Az IH egy G bemenetéből a leképezéssel olcsón kaphatjuk a H feladat G' bemenetét. Ha a ($G' \in$ H?) kérdést meg tudjuk válaszolni, akkor megkaptuk a választ a ($G \in$ IH?) kérdésre is. Most pedig nézzük mindezt általánosabban.

Állítás:
1. Ha $L_1 \prec L_2$ *és* $L_2 \in$ P, *akkor* $L_1 \in$ P.
2. Ha $L_1 \prec L_2$ *és* $L_2 \in$ NP *akkor* $L_1 \in$ NP.
3. Ha $L_1 \prec L_2$, *akkor* $\overline{L}_1 \prec \overline{L}_2$, *ahol* $\overline{L}_i = I^* \setminus L_i$.
4. Ha $L_1 \prec L_2$ *és* $L_2 \in$ *co*NP, *akkor* $L_1 \in$ *co*NP.
5. Ha $L_1 \prec L_2$ *és* $L_2 \in$ NP $\cap$ *co*NP, *akkor* $L_1 \in$ NP $\cap$ *co*NP.

Bizonyítás: Legyen $f : I^* \rightarrow I^*$ az L_1 Karp-redukciója L_2-re. A definíció szerint $f \in$ FP, mondjuk $f \in FTIME(n^k)$. Jelöljön az $x \in I^*$ egy input szót, melyre szeretnénk eldönteni, hogy $x \in L_1$ teljesül-e, és legyen n az x hossza. Vegyük ezután sorra az állításokat.
1. A célunk itt az, hogy polinom idejű algoritmust adjunk az L_1 felismerésére. Az f függvény mellett még használhatjuk az $L_2 \in$ P feltételt, mondjuk legyen $L_2 \in TIME(n^l)$. Először kiszámítjuk az $f(x)$ szót, ennek időigénye legfeljebb $c_1 n^k$, amiből $|f(x)| \leq c_1 n^k$ is következik. A második lépésben L_2 felismerő algoritmusával eldöntjük, hogy $f(x) \in L_2$ igaz-e. Ennek az időigénye legfeljebb $c_2(c_1 n^k)^l$. A Karp-redukció 1. tulajdonsága szerint az ($x \in L_1$?) kérdésre adandó válasz megegyezik az ($f(x) \in L_2$?) kérdésre adott válasszal, tehát az x inputot pontosan akkor fogadjuk el, ha a második lépésben $f(x)$-et elfogadtuk. A számítás összideje $O(n^{kl})$. Ezzel igazoltuk, hogy $L_1 \in$ P.
2. Az $f(x) \in L_2$ tény egy y tanúja megfelelő lesz az $x \in L_1$ tanújának is, és az L_2-höz tartozó bíró egy kis módosítással jó lesz az L_1 bírójának is. Ha ugyanis

$|y| \le |f(x)|^c$, akkor $|y| \le c_1^c|x|^{kc}$, tehát y az x-hez képest polinomiális méretű. Az L_1 bírója az (x, y) bemenet esetén először kiszámítja $f(x)$-et, majd átadja az $(f(x), y)$ párt az L_2 bírájának. Az L_1 bírója pontosan akkor fogadja el az (x, y) párt, ha az L_2 bírója elfogadja az $(f(x), y)$ párt. Ez a Karp-redukció definíciója szerint éppen azt jelenti, hogy $x \in L_1$. Az is világos, hogy L_1 bírójának az időigénye becsülhető n egy polinomjával.
3. Az $L_1 \prec L_2$ redukciót megvalósító f jó, hiszen a definíció 1. követelménye így is fogalmazható: tetszőleges $x \in I^*$ szóra $x \notin L_1$ pontosan akkor teljesül, ha $f(x) \notin L_2$.
4. Közvetlenül adódik az előző két állításból.
5. A 2. és a 4. állítások következménye. □

A megelőző állítások szerint ha L_2 benne van valamelyik nevezetes P-t tartalmazó nyelvosztályban, akkor L_1 is. Ez érvényes a PSPACE és EXPTIME osztályokra is; az első állítás gondolatmenete működik ezekben az esetekben is.

Feladat: Igazoljuk, hogy a $\prec$ reláció tranzitív: ha $L_1 \prec L_2$ és $L_2 \prec L_3$, akkor $L_1 \prec L_3$ is teljesül.

A Karp-redukció fogalma lehetőséget ad arra, hogy könnyebb-nehezebb viszonyt állapítsunk meg nyelvek között. Az NP osztály egyik érdekes tulajdonsága, hogy vannak benne ilyen értelemben *legnehezebb* nyelvek (feladatok). Ezek az NP-*teljes nyelvek*.

Definíció: *Az $L \subseteq I^*$ nyelv* NP-*teljes, ha*
1. $L \in$ NP,
2. tetszőleges $L' \in$ NP *nyelv esetén létezik $L' \prec L$ Karp-redukció.*

Egy NP-teljes nyelv tehát legalább olyan nehéz, mint bármely más NP-beli nyelv. Ha egy ilyen nyelvről kiderülne, hogy P-beli (coNP-beli), akkor ugyanez igaz lenne minden NP-beli nyelvre. Az NP-beli nyelveknek és a megfelelő (tanú)keresési feladatoknak a kapcsolata miatt az NP-teljes nyelveket szokás még *univerzális kereső feladatoknak* is nevezni. A P $\neq$ NP sejtés szellemében úgy gondoljuk, hogy az NP-teljes nyelvekre nem léteznek polinom idejű felismerő módszerek. Ez nem bizonyított tény, de óriási mennyiségű közvetett bizonyíték szól mellette: a gyors algoritmusokat kereső kísérletek egyhangú kudarca. Ilyen értelemben az NP-teljes feladatok nehézségét a tudomány mai állása mellett *tapasztalati ténynek* tekintjük. Azt is hangsúlyozzuk, hogy az NP-teljes feladatok nem megoldhatatlanok. Korábban láttuk, hogy

$$\text{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME},$$

tehát az NP-beli problémák legnehezebbjei is legyűrhetők exponenciális idejűnél nem rosszabb algoritmussal.

8.6. A SAT nyelv és a Cook–Levin-tétel

Fiam, tanácsolom tehát,
próbáld először a logikát.
...
hogy ezután már ne ingatag
járja útját a gondolat,
fel s le bolyongva botorul,
lidércként keresztül-kasul.

JOHANN WOLFGANG GOETHE
(Mefisztó tanácsai a Diáknak.)

A következőkben egy alapvető fontosságú NP-teljes nyelvet ismertetünk. Szükségünk lesz ehhez a *Boole-formula* fogalmára. Egy Boole-formula a 0 („hamis"), 1 („igaz") logikai konstansokból, 0-1 értékű változókból (mondjuk ezek $x_1, \ldots, x_n$) és a változók $\overline{x}_1, \ldots, \overline{x}_n$ negáltjaiból a $\wedge$ („és"), illetve a $\vee$ („vagy") műveleti jelekkel és zárójelekkel felépített kifejezés. Szokás a változókat és negáltjaikat közös névvel illetve *literál*oknak nevezni. Egy Boole-formula változóinak (logikai) értékeket adhatunk. A változók ilyen *kiértékelése* után a kézenfekvő módon beszélhetünk a formula értékéről, ami 0 vagy 1 lesz. A ϕ Boole-formula *kielégíthető*, ha van a változóinak olyan kiértékelése, melynél ϕ értéke 1.

Példa: Az $(x_1 \vee x_2) \wedge \overline{x}_3$ formula kielégíthető ($x_1 = 1$, $x_2 =$ értéke tetszőleges, $x_3 = 0$). Az $x_1 \wedge \overline{x}_1$ formula pedig nem kielégíthető.

Jelölje SAT a kielégíthető Boole-formulák nyelvét (a jelölés a *kielégíthetőség* szó angol megfelelőjéből, a *satisfiability* szóból származik). Ebben az esetben sem túl fontos, hogy miként kódoljuk a formulákat bináris szavakká. Egyik lehetséges eljárás, hogy a formula elemeinek (változók, állandók, műveleti jelek, zárójelek) kódjait fűzzük össze a kézenfekvő módon. A SAT nyelv benne van az NP osztályban: egy formula kielégíthetőségének tanúja egy jó kiértékelés. A bíró kiszámolja a formula értékét az adott kiértékelésnél. A következő eredmény szerint a SAT a legnehezebbek egyike az NP osztályban.

Tétel (S. A. Cook, L. Levin, 1971): *A SAT egy* NP-*teljes nyelv.*

Bizonyítás: (Vázlat) A SAT $\in$ NP állítással már foglalkoztunk. A tétel érdemi részéhez azt kell igazolni, hogy tetszőleges $L \in$ NP nyelvre létezik egy $L \prec$ SAT Karp-redukció.

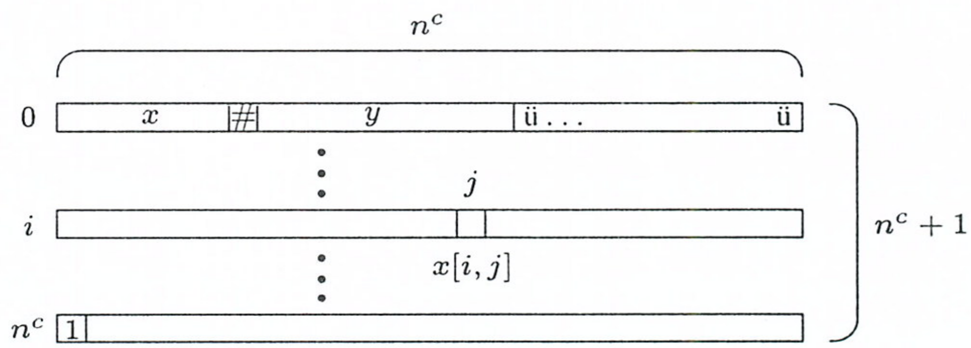
Mit jelent egy ilyen redukció? Az $(x \in L?)$ kérdés tetszőleges $x \in I^*$ inputjához meg kell adnunk egy ϕ Boole-formulát, mely pontosan akkor kielégíthető, ha

$x \in L$. Ennek az $x \mapsto \phi$ leképezésnek hatékonyan (azaz polinom időben) számíthatónak kell lennie. Egyszerűbben fogalmazva: az x inputra a ϕ formulát meg kell tudnunk adni $O(|x|^k)$ lépésben.

A tanú-tétel szerint az $L \in$ NP feltétel azt jelenti, hogy van olyan $L_1 \in$ P nyelv (a bíró nyelve) és $d > 0$ konstans, hogy

$$x \in L \text{ pontosan akkor teljesül,}$$
$$\text{ha van olyan } y \in I^*, \text{ melyre } |y| \leq |x|^d, \text{ és } (x, y) \in L_1.$$

Legyen M egyszalagos, polinomkorlátos TG, mely az L_1 nyelvet ismeri fel. Az egyszerűség kedvéért feltesszük még, hogy M szalagjelei 0,1 és ü. Tegyük fel, hogy az (x, y) összetett bemenet az M szalagján $x\#y$ alakban van, ahol $\#$ egy elválasztó bitsorozat. Jelölje n az x input szó hosszát, és tegyük fel, hogy $n^{1+d} + |\#|$ hosszúságú inputokon M legfeljebb n^c lépést tesz. Ilyen c kitevő létezik, mert M polinomkorlátos. Esetleg az M egy kis módosítása árán még azt is feltehetjük, hogy az M pontosan akkor fogadja el az inputot, ha megállás után a szalag első bitje 1 (ha elfogadó állapotba kerül, 1-et ír a szalag elejére). Az M működését az $x\#y$ inputon az alábbi tártérkép mutatja. Az i-edik sor M szalagját ábrázolja az i-edik lépés megtétele utáni helyzetben.



A ϕ Boole-formulával kapcsolatos követelményeket úgy is fogalmazhatjuk, hogy ϕ hatékonyan számítható x-ből, és pontosan akkor kielégíthető, ha van olyan rövid y, melyre M az $x\#y$ inputot elfogadja. A ϕ konstrukciójához szükségünk lesz a gép pillanatnyi helyzeteit leíró 0-1 értékű változókra: ($i = 0, \ldots, n^c$; $j = 1, \ldots, n^c$; $s = 0, 1, \ldots, |Q| - 1$)

$$0x[i,j] = \begin{cases} 1 & \text{ha az } i\text{-edik lépés után a } j\text{-edik cella tartalma 0} \\ 0 & \text{különben} \end{cases}$$

$$1x[i,j] = \begin{cases} 1 & \text{ha az } i\text{-edik lépés után a } j\text{-edik cella tartalma 1} \\ 0 & \text{különben} \end{cases}$$

$$\text{ü}x[i,j] = \begin{cases} 1 & \text{ha az } i\text{-edik lépés után a } j\text{-edik cella tartalma ü} \\ 0 & \text{különben} \end{cases}$$

$$f[i,j] = \begin{cases} 1 & \text{ha az } i\text{-edik lépés után a fej } j\text{-edik cellán áll} \\ 0 & \text{különben} \end{cases}$$

$$q[i,s] = \begin{cases} 1 & \text{ha az } i\text{-edik lépés után } M \text{ belső állapota } q_s \\ 0 & \text{különben.} \end{cases}$$

Az M működését szeretnénk leírni ezekből a változókból épített Boole-formulákkal. Többféle tényt kell ilyen módon megfogalmaznunk:

1. Le kell írnunk olyan állításokat, hogy a szalag egy mezőjén minden időpontban éppen egy szalagjel van; a fej minden időpontban a szalag egyetlen celláján van; a gép minden időpontban egyetlen állapotban van. Példaként formalizáljuk az első típusba eső, a szalagjelekkel foglalkozó állításokat: minden i, j ($0 \leq i \leq n^c$, $1 \leq j \leq n^c$) párra

$$(0x[i,j] \vee 1x[i,j] \vee \text{ü}x[i,j]) \wedge$$
$$\wedge(\overline{0x[i,j]} \vee \overline{1x[i,j]}) \wedge (\overline{0x[i,j]} \vee \overline{\text{ü}x[i,j]}) \wedge (\overline{1x[i,j]} \vee \overline{\text{ü}x[i,j]}).$$

Összesen $(n^c + 1)n^c$ ilyen formulánk lesz. Hasonlóan fogalmazhatjuk meg a fej helyzetével és a belső állapottal kapcsolatos állításokat.

2. Le kell írnunk a gép átmeneteit is (lényegében a δ függvényt). Ezt is példával mutatjuk be. Tegyük fel, hogy a q_s belső állapotra és az 1 szalagjelre $\delta(q_s, 1) = (q_k, 0, bal)$ (M a q_s állapotból 1-es szalagjel olvasása esetén a q_k állapotba megy át, 0-t ír a fej alatti cellába, majd a fej balra lép). Ezt a változóinkkal így fejezhetjük ki: tetszőleges $0 \leq i < n^c$, $1 \leq l \leq n^c$ esetén

$$((q[i,s] \wedge 1x[i,l] \wedge f[i,l]) \longrightarrow (q[i+1,k] \wedge 0x[i+1,l] \wedge f[i+1,l-1]).$$

Emlékeztetőül: az $u \to v$ (ha u igaz, akkor v is igaz) kifejezés az $\overline{u} \vee v$ Boole-formula egy másik írásmódja. Összesen $O(n^{2c})$ ilyen formulánk lesz, mert a δ táblázat mérete független n-től.

Az előbbi formulák leírják, hogy mi és hogyan változik. Azt is meg kell mondanunk, hogy más változás nincs; a szalag azon cellái, amelyek felett *nincs* a fej, megtartják régi értéküket.

$$\overline{f[i,l]} \longrightarrow (0x[i,l] \leftrightarrow 0x[i+1,l])$$

Ugyanilyen formulák kellenek még az 1 és az ü jelekkel is. Ezek együttes száma szintén $O(n^{2c})$.

3. A gép kezdő helyzetének (a q_0 belső állapotban van, a fej az első cellára mutat) a leírása így néz ki:

$$q[0,0] \wedge f[0,1]$$

Legyen ϕ_m az m-edik lépés utáni helyzetet leíró formulák $\wedge$-e (konjunkciója). A $\psi = \phi_0 \wedge \ldots \wedge \phi_{n^c}$ formula azt fejezi ki, hogy a tártérkép a 0. sor alatt úgy van kitöltve, ahogy azt M tenné.

Jelölje $\psi(x)$ azt a formulát, melyet úgy kapunk ψ-ből, hogy $\wedge$-el hozzákapcsoljuk azt az állítást, mely szerint az input # előtti része éppen az x szóval egyezik meg (pl. $\psi(x) = \psi \wedge 0x[0,1] \wedge 1x[0,2] \wedge 0x[0,3] \ldots$, ha $x = 010\ldots$). Végül legyen

$$\phi = 1x[n^c, 1] \wedge \psi(x).$$

Az $f[i,j], q[i,s], 0x[i,j], 1x[i,j], \ddot{u}x[i,j]$ változók valamely kiértékelésénél a ϕ formula értéke pontosan akkor lesz igaz, ha van M-nek olyan legfeljebb n^c lépéses elfogadó számítása, melynél az input első fele x. A ϕ kiértékelésekor a tényleges szabadsági fok az y súgásnak megfelelő $0x[0,j], 1x[0,j], \ddot{u}x[0,j]$ változók értékadásában van. A ϕ tehát pontosan akkor lesz kielégíthető formula, ha az inputnak a # utáni része kitölthető úgy, hogy ebből a helyzetből indulva M elfogadó állapotban álljon meg a munkája végén. Ez pedig éppen azt jelenti, hogy $x \in L$. A vázolt módon a ϕ formula x és M ismeretében polinom időben megadható, tehát a kapott $x \mapsto \phi$ függvény tényleg egy $L \prec$ SAT Karp-redukció. □

A bizonyítás szerint a gép működését le lehet írni egyetlen Boole-formulával. Eme figyelemre méltó tény a logika kifejező erejét mutatja. A logikának ez a nagyon erős leíró képessége fontos szerepet játszik a számítástudomány más területein is; említhetjük itt a logikai programozás nyelveinek hajlékonyságát, vagy a modern adatbázisok lekérdező felületeinek (mint pl. az SQL) gazdag és emberi léptékű kifejező erejét.

8.7. További NP-teljes feladatok

Ebben a részben további NP-teljes feladatokkal ismerkedünk meg. Felvetődik a kérdés, hogy miért foglalkozunk ezekkel ilyen behatóan, ha ezekre – széles körben elfogadott sejtés szerint – nem adható hatékony algoritmus. Azért szentelünk ennyi figyelmet az NP-teljes feladatoknak, mert szinte mindenütt ott vannak. Tapasztalati tény, hogy a fontos, érdekes algoritmikus problémák között igen gyakran találunk ilyeneket (a szakirodalomban dokumentált NP-teljes feladatok száma jóval ezer felett van). Az is gyakori jelenség, hogy egy kezelhető (mondjuk P-beli) probléma paramétereinek látszatra ártatlan változtatásával már NP-teljes feladatot kapunk. Ha egy problémáról tudjuk, hogy NP-teljes, akkor azt is tudjuk, hogy

nem érdemes sokat fáradozni hatékony algoritmus keresésén. Néhány lehetséges kerülőúttal a következő fejezetben foglalkozunk. Az NP-teljességgel kapcsolatos ismeretek tehát segítenek eligazodni abban, hogy egy adott algoritmikus feladatra milyen (mennyire gyors, milyen pontos) megoldás keresése lehet reális célkitűzés.

Az alábbi egyszerű állítás segítségével igazolhatjuk nyelvek NP-teljességét.

Állítás: *Ha az* L_1 *nyelv* NP*-teljes,* $L_2 \in \text{NP}$ *és* $L_1 \prec L_2$*, akkor* L_2 *is* NP*-teljes.*

Bizonyítás: Mivel $L_2 \in \text{NP}$, elég megmutatni, hogy $L' \prec L_2$ minden NP-beli L' nyelv esetén. A feltételek szerint $L' \prec L_1$ és $L_1 \prec L_2$. Legyenek f és g a redukciókat megvalósító FP-beli függvények. Ekkor $g \circ f$ (az a leképezés, mely az x szóhoz a $g(f(x))$ szót rendeli) egy $L' \prec L_2$ Karp-redukció. Teljesül tehát a definíció második követelménye is, így L_2 tényleg NP-teljes. □

A Cook–Levin-tétel után szinte minden NP-teljességet bizonyító gondolatmenet az előző állításból adódó utat használja. Nem kell már *minden* NP-beli nyelvet az L_2-re redukálni; elég ezt megtenni *egyetlen* NP-teljes L_1 nyelvvel.

Ha az L_2 nyelvről csak azt tudjuk, hogy van olyan NP-teljes L_1 nyelv, melyre $L_1 \prec L_2$, akkor L_2-t NP-*nehéz* nyelvnek nevezzük. Az előző állítás szerint L_2 pontosan akkor NP-teljes, ha NP-beli és ugyanakkor NP-nehéz is.

8.7.1. Konjunktív normálformájú formulák kielégíthetősége és a 3-SAT

A ϕ Boole-formula *konjunktív normálformájú*, ha $\phi = \phi_1 \wedge \cdots \wedge \phi_k$ alakú, ahol a ϕ_i formulákban literálok szerepelnek kizárólag $\vee$ jelekkel összekapcsolva. A ϕ_i formulákat ekkor a ϕ *tagjainak* nevezzük. Az ítéletkalkulus azonosságai (disztributivitás, De Morgan szabályok) segítségével könnyen látható, hogy minden formulához van egy vele ekvivalens (minden kiértékelésre ugyanazt az értéket adó) konjunktív normálformájú formula.

Feladat: Mutassuk meg, hogy egy Boole-formula konjunktív normálformájú ekvivalensének kiszámítására nincs polinom idejű algoritmus. (Előfordulhat, hogy az eredmény mérete nem polinomiális az input méretéhez képest.)

Ha a ϕ_i formulákban legfeljebb k literál szerepel, akkor ϕ-t k*-konjunktív normálformájú* formulának (szokásos rövidítéssel k-CNF-nek) nevezzük.

Példa: Az alábbi formula egy 4-CNF:

$$(\overline{x}_1 \vee x_2 \vee x_3 \vee x_4) \wedge (x_4 \vee \overline{x}_7 \vee \overline{x}_{18}) \wedge (x_9 \vee \overline{x}_{10} \vee \overline{x}_{13}).$$

Jelölje k-SAT a kielégíthető k-CNF-ekből álló nyelvet. Egy k-CNF kielégíthetőségének a tanúja lehet egy kielégítő kiértékelés. A tanú a formula értékének kiszámításával hatékonyan ellenőrizhető, tehát k-SAT $\in$ NP. Ez a nyelv könnyű, ha $k = 2$:

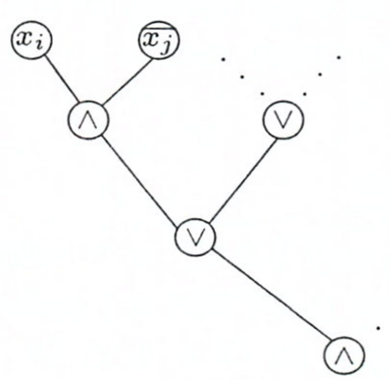
Feladat: Mutassuk meg, hogy 2-SAT $\in$ P. (Legyen ϕ egy 2-CNF. Jelölje ϕ_0, illetve ϕ_1 azokat a 2-CNF-eket amelyeket ϕ-ből az $x_1 = 0$, illetve $x_1 = 1$ helyettesítések után kaptunk. A ϕ_i-ről gyorsan kideríthetjük a következők *valamelyikét*:
1. ϕ_i kielégíthető,
2. ϕ_i nem kielégíthető,
3. ϕ_i kielégíthetősége ekvivalens egy ϕ_i' 2-CNF kielégíthetőségével, aminek tagjai ϕ-nek is tagjai, és amelyben az x_1 változó nem szerepel.)

Ha eggyel több literált engedünk meg a zárójelekben, akkor a helyzet alaposan megváltozik; a könnyű problémából nehéz lesz.

Tétel: *A 3-SAT nyelv* NP-*teljes.*

Bizonyítás: Az előzőek alapján elég megmutatni, hogy SAT $\prec$ 3-SAT. Olyan polinom időben számítható megfeleltetést kell adnunk, mely egy ϕ formulához (a SAT bemenetéhez) egy ψ 3-CNF-et rendel úgy, hogy ϕ akkor és csak akkor kielégíthető, ha ψ kielégíthető.

Legyenek $x_1, x_2, \ldots, x_n$ a ϕ formulában szereplő változók, és írjuk fel a ϕ egy kifejezésfáját, majd számozzuk meg a fa csúcsait, minden csúcshoz egyedi sorszámot rendelve. A fa csúcsai a ϕ részformuláinak felelnek meg. A csúcs feletti leveleknek értéket adva a részformula kiszámítható; e megfeleltetés révén beszélhetünk a csúcshoz tartozó értékről. Az i sorszámú csúcshoz a (ϕ változóitól és egymástól is különböző) z_i Boole-változót rendeljük.



Ezen felül az i sorszámú csúcshoz rendelünk még egy ϕ_i Boole-formulát. Egyszerű szavakba foglalva: a ϕ_i formula azt fogja kifejezni, hogy a z_i értéke legyen

az i-edik csúcshoz tartozó érték az x_j változók egy adott kiértékelésekor. Innen a ϕ_i formulák definíciója már kézenfekvő (az $u \leftrightarrow v$ kifejezés az $(\overline{u} \vee v) \wedge (u \vee \overline{v})$, azaz a logikai ekvivalencia rövidítése):
Ha az i-edik csúcs levél, amelynek tartalma az u literál (x_j, vagy $\overline{x}_j$), akkor legyen $\phi_i : \; z_i \leftrightarrow u$.
Ha az i-edik csúcsban $\wedge$ van, és a fiai a k-adik és l-edik csúcsok, akkor legyen $\phi_i : \; z_i \leftrightarrow (z_k \wedge z_l)$.
Ha az i-edik csúcsban $\vee$ van, és a fiai a k-adik és l-edik csúcsok, akkor legyen $\phi_i : \; z_i \leftrightarrow (z_k \vee z_l)$.

Jelölje m a kifejezésfa gyökerének a sorszámát. Következő lépésként vegyünk mindegyik ϕ_i-hez egy ekvivalens konjunktív normálformájú formulát. Legyen ez ψ_i. A ϕ_i formulában legfeljebb 3 változó van (bizonyos x-ek és bizonyos z-k), ezért ψ_i egy 3-CNF. Legyen $\psi = (\wedge\psi_i) \wedge z_m$. A ψ egy 3-CNF, lévén ilyenek konjunkciója. A ψ értéke a változóinak a kiértékelésekor pontosan akkor lesz 1 (vagyis „igaz"), ha ennél a kiértékelésnél mindegyik ψ_i és z_m is igaz. Mindez egyenértékű azzal, hogy minden i-re a z_i változó értéke megegyezik a ϕ kifejezésfa szerinti számításánál az i-edik csúcshoz tartozó részeredménnyel, és (a z_m tag jelenléte miatt) a ϕ értéke is 1. Úgy is fogalmazhatunk, hogy az x_i értékek tetszőleges megválasztása után a ψ_i formulák egyértelműen igazzá tehetők: a z_i változónak az i-edik csúcsnál fellépő értéket kell adni. A ψ ezek szerint pontosan akkor kielégíthető, ha ϕ is ilyen tulajdonságú. Más szóval $\phi \in$ SAT pontosan akkor teljesül, ha $\psi \in$ 3-SAT. Végül megjegyezzük, hogy a ψ előállítására adott recept RAM-on lineáris időben implementálható. A $\phi \mapsto \psi$ megfeleltetés tényleg egy Karp-redukció. □

8.7.2. 3 színnel színezhető gráfok

Korábban már láttuk, hogy a 3-SZÍN, a három színnel színezhető gráfok nyelve az NP osztályban van. A következő állítás szerint ez a nyelv is az NP legnehezebbjei közül való.

Tétel: *A 3-SZÍN nyelv* NP-*teljes.*

Bizonyítás: Elegendő megadni egy 3-SAT $\prec$ 3-SZÍN Karp-redukciót. Pontosabban fogalmazva, egy tetszőleges ψ 3-CNF-hez építenünk kell egy G gráfot úgy, hogy $\psi \in$ 3-SAT pont akkor igaz, ha $G \in$ 3-SZÍN. A konstrukciónak hatékonynak, polinom időben elvégezhetőnek kell lennie.

Tegyük fel, hogy a ψ formulában m változó és z nyitózárójel van: $\psi = (\psi_1) \wedge \ldots \wedge (\psi_z)$. Feltehető, hogy minden egyes ψ_i pontosan 3 tagú (például $x_1 \vee x_2$ helyett vehetjük az $x_1 \vee x_2 \vee x_2$ formulát). A G gráfnak $2m + 5z + 2$

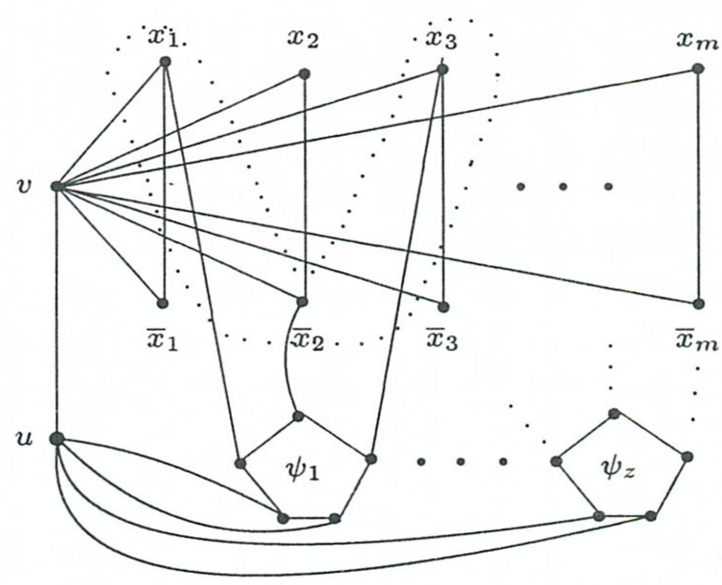
csúcsa lesz. Van két különálló csúcs: u és v. A ψ-beli x_i változónak és negáltjának is megfelel egy-egy csúcs; ezek jele x_i, illetve $\overline{x}_i$. Végül a ψ_j részformulának 5 csúcs fog megfelelni; ezek $\psi_j^1, \ldots, \psi_j^5$.

$$V(G) = \cup_{i=1}^{m}\{x_i, \overline{x}_i\} \cup \cup_{j=1}^{z}\{\psi_j^1, \ldots, \psi_j^5\} \cup \{v, u\}.$$

A G élei a következők:

$$E(G) = \left\{ \begin{array}{ll} (v,u), & \\ (v,x_i), & i = 1,\ldots,m, \\ (v,\overline{x}_i), & i = 1,\ldots,m, \\ (x_i,\overline{x}_i), & i = 1,\ldots,m, \\ (u,\psi_j^4), & j = 1,\ldots,z, \\ (u,\psi_j^5), & j = 1,\ldots,z, \\ (\psi_j^k,\psi_j^{k+1}), & j = 1,\ldots,z,\ k \text{ modulo 5 fut}, \\ (\psi_j^k,x_i), & \text{ahol } \psi_j\ k\text{-adik tagja } x_i,\ j = 1,\ldots,z,\ k = 1,2,3, \\ (\psi_j^k,\overline{x}_i), & \text{ahol } \psi_j\ k\text{-adik tagja } \overline{x}_i,\ j = 1,\ldots,z,\ k = 1,2,3. \end{array} \right\}$$

A G gráfban ötszögek felelnek meg a ψ_j formuláknak. Ezek „alsó" pontjai u-val vannak összekötve, a felső pontjai pedig egy-egy a ψ_j-ben előforduló literállal. A következő ábra azt az esetet szemlélteti, amikor $\psi_1 = x_1 \vee \overline{x}_2 \vee x_3$.



A G mérete lineáris ψ méretében, és az is látható, hogy G könnyen (polinom időben) felépíthető ψ ismeretében. Igazolnunk kell még, hogy

$$G \in \text{3-SZÍN pontosan akkor, ha } \psi \in \text{3-SAT}.$$

Vegyük először G egy 3-színezését. Nevezzük v színét pirosnak, u-ét sárgának, a harmadik szín legyen a zöld. Vegyük észre, hogy egy literál színe csak zöld vagy sárga lehet a v-vel összekötő él miatt. A ψ formula y literáljának adjuk az *igaz* értéket, ha a G-ben az y csúcs zöld; ha y sárga színű, akkor az értéke legyen *hamis*. Ez megfelel az x_i változók egy kiértékelésének. Az $(x_i, \overline{x}_i)$ él megléte miatt az x_i és $\overline{x}_i$ csúcsok közül az egyik sárga, a másik zöld. Megmutatjuk, hogy így a ψ egy kielégítő kiértékelése adódik. Ez egyenértékű azzal az állítással, hogy mindegyik ψ_i-ben van zöld literál. Ez pedig igaz, hiszen ellenkező esetben a ψ_i-hez tartozó ötszög csúcsaira csak két szín volna megengedett (piros és zöld); az ötszög viszont nem színezhető két színnel.

A fordított irányú állításhoz vegyük ψ egy kielégítő kiértékelését. Feleltessük meg, mint előbb, a literálok *igaz* értékének a zöld színt, a *hamis* értéknek pedig a sárga színt. Színezzük v-t pirosra, u-t pedig sárgára. Az eddig kapott részleges színezéssel nincs baj, már csak a ψ_i formulákhoz tartozó ötszögeket kell kiszínezni. Egy ilyen ötszög minden csúcsára tiltott a zöld és a sárga színek egyike. Az alsó két pontra a sárga, a felső csúcsokra viszont nem mindenütt a sárga, hiszen ψ_i-ben van zöld literál. Az alábbi egyszerű feladat szerint ilyenkor az ötszög kiszínezhető. □

Feladat: Az ötpontú kör kiszínezhető 3 színnel úgy is, hogy minden egyes csúcsára kijelölünk egy tiltott színt, de nem minden csúcsra ugyanazt a színt.

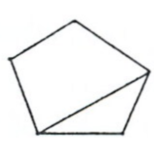
Feladat: Adjunk lineáris (uniform) költségű algoritmust egy gráf két színnel való színezhetőségének eldöntésére.

8.7.3. Maximális méretű független pontrendszer gráfokban

A G gráf csúcsainak S részhalmaza *független halmaz*, ha S-beli pontok között nem fut G-ben él. A G gráf fontos jellemzője a legnagyobb méretű független ponthalmazának elemszáma. Ennek a kiszámítása nehéz algoritmikus probléma, amint azt a MAXFTLEN nyelv mutatja.

$$\text{MAXFTLEN} = \left\{ (G,k) \;\middle|\; \begin{array}{l} G \text{ egy gráf, } k \in \mathbb{Z}^+, \text{ és} \\ G\text{-nek van } k \text{ elemű független csúcshalmaza} \end{array} \right\}.$$

Példa: Az ábrán látható G gráf esetén $(G,1), (G,2) \in$ MAXFTLEN, $(G,3) \notin$ MAXFTLEN.

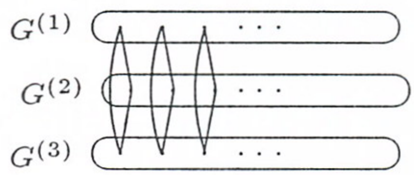


Tétel: *A MAXFTLEN nyelv* NP-*teljes.*

Bizonyítás: Nyilvánvaló, hogy MAXFTLEN $\in$ NP: a $(G, k) \in$ MAXFTLEN ténynek tanúja lehet egy k-elemű $S \subseteq V(G)$ független csúcshalmaz. Elegendő ezután megadni egy 3-SZÍN $\prec$ MAXFTLEN Karp-redukciót. Legyen a G gráf a 3-SZÍN egy inputja. Olyan (G_1, k) párt kell hatékonyan megadni (G_1 egy gráf, k egy pozitív egész), melyre érvényes a következő:

$$G \in \text{3-SZÍN pontosan akkor, ha } (G_1, k) \in \text{MAXFTLEN}. \qquad (*)$$

A G_1 gráfot a következőképpen kapjuk a G-ből: veszünk három példányt G-ből, ezek legyenek $G^{(1)}, G^{(2)}, G^{(3)}$, és kössük össze élekkel a G azonos csúcsainak megfelelő pontokat. Így a $v \in V(G)$ csúcs három példánya $v^{(i)} \in G^{(i)}$, $(i = 1, 2, 3)$ egy háromszöget feszít G_1-ben.



A G_1 csúcsainak és éleinek számára a következőket kapjuk:

$$|V(G_1)| = 3|V(G)| \text{ és } |E(G_1)| = 3|V(G)| + 3|E(G)|.$$

Legyen $k = |V(G)|$. A G ismeretében a (G_1, k) pár hatékonyan megadható. A tétel bizonyításához ezután elegendő a $(*)$ állítást igazolni.

Tekintsük először a G egy 3 színnel való színezését, ahol a színek 1,2,3. Álljon a $H \subseteq V(G_1)$ halmaz azokból a $v^{(i)}$ pontokból, melyekre $v \in V(G)$ színe i. Másképp fogalmazva: egy i színű G-beli pontnak az i-edik szinten levő példányát vesszük H-ba. A színezés definíciójából azonnal kapjuk, hogy a H halmaz független, és $|H| = k$. Van tehát G_1-ben k független pontból álló halmaz.

Fordítva, legyen H egy k elemű független halmaz G_1-ben. Nyilvánvaló, hogy ekkor H minden oszlopból ($v^{(1)}, v^{(2)}, v^{(3)}$ hármasból) pontosan egy csúcsot tartalmaz. Legyen a $v \in V(G)$ csúcs színe i, ha $v^{(i)} \in H$. Ez jó színezése lesz G-nek, hiszen ha az u és v csúcsok azonos színűek, akkor $u^{(i)}, v^{(i)} \in H$ valamely i-re. Ekkor pedig H függetlensége miatt $(u, v) \notin E(G)$. □

A MAXFTLEN nyelvre vonatkozó tételből könnyen adódik két rokon gráfelméleti probléma NP-teljessége. Legyen

$$\text{MAXKLIKK} = \{(G, k) |\ G\text{-ben van } k \text{ pontú teljes részgráf}\}.$$

Következmény: *A MAXKLIKK nyelv* NP*-teljes.*

Bizonyítás: Nyilvánvaló, hogy MAXKLIKK $\in$ NP. Az $X \subseteq V(G)$ csúcshalmaz pontosan akkor független a G gráfban, ha X klikk (teljes részgráfot feszít) a G gráf $\overline{G}$ komplementer gráfjában. (A $\overline{G}$ pontjai ugyanazok, mint G pontjai, és (u, v) él $\overline{G}$-ben pontosan akkor, ha (u, v) nem él G-ben.) Ebből azonnal adódik, hogy a $(G, k) \mapsto (\overline{G}, k)$ megfeleltetés egy MAXFTLEN $\prec$ MAXKLIKK Karp-redukció. □

A G gráf csúcsainak egy X részhalmaza *éllefogó halmaz*, ha a G bármely élének van X-beli végpontja. Legyen

$$\textit{Éllefogás} = \{(G, k) |\ G\text{-ben van } k \text{ pontú éllefogó halmaz}\}.$$

Feladat: Mutassuk meg, hogy a $H \subseteq V(G)$ halmaz akkor, és csak akkor független halmaz G-ben, ha $V(G) \setminus H$ egy éllefogó halmaz.

Következmény: *Az Éllefogás nyelv* NP*-teljes.*

Bizonyítás: Nyilvánvaló, hogy *Éllefogás* $\in$ NP. Másfelől az előző feladatból azonnal következik, hogy a $(G, k) \mapsto (G, |V(G)| - k)$ megfeleltetés egy MAXFTLEN $\prec$ *Éllefogás* Karp-redukció. □

Megjegyezzük, hogy a MAXFTLEN, 3-SZÍN és *Éllefogás* nyelvek akkor is NP-teljesek maradnak, ha még azt is kikötjük, hogy G egy síkba rajzolható gráf.

Feladat: Mutassuk meg, hogy síkba rajzolható gráfokra a MAXKLIKK feladat megoldható polinom időben.

Feladat: Mutassuk meg, hogy páros gráfokra az *Éllefogás* feladat polinom időben megoldható. (König Dénes minimax tétele.)

8.7.4. A 3 dimenziós házasítás és az X3C feladat

A következő algoritmikus probléma a korábban megismert párosítási feladat általánosításának tekinthető. Legyenek U_1, U_2, U_3 azonos méretű véges halmazok, mondjuk legyen $|U_i| = t$. Tegyük fel, hogy adott még $U_1 \times U_2 \times U_3$ valamely S részhalmaza. Egy ilyen részhalmaz (u_1, u_2, u_3) alakú hármasokból áll, ahol $u_i \in U_i$. Szeretnénk eldönteni, hogy kiválasztható-e S-ből egy *3 dimenziós házasítás*. Ezen az S egy olyan t-elemű S' részhalmazát értjük, mely minden U_i-beli pontot lefed. Utóbbi követelmény pontosabban így fogalmazható: tetszőleges $v_i \in U_i$ elemhez van olyan $s \in S'$, melynek i-edik komponense v_i. Az $|S'| = t$

feltétel miatt egy 3 dimenziós házasítás az $U_1 \cup U_2 \cup U_3$ elemeit pontosan egyszeresen fedi le.

Jelölje 3-DH az olyan $U_1, U_2, U_3;\ S \subseteq U_1 \times U_2 \times U_3$ rendszereket, melyeknél S-ből kiválasztható egy 3 dimenziós házasítás.

Tétel: *A 3-DH feladat* NP-*teljes.*

Bizonyítás: Nyilvánvaló, hogy 3-DH $\in$ NP. Egy alkalmas S' halmaz játszhatja a tanú szerepét. A bizonyítás érdemi része egy 3-SAT $\prec$ 3-DH redukció lesz. Legyen $\phi \in$ 3-CNF. Tegyük fel, hogy ϕ-ben m változó $(x_1, \ldots, x_m)$ és z nyitózárójel van: $\phi = (\phi_1) \wedge \ldots \wedge (\phi_z)$.

Az U_1 halmazt úgy képezzük, hogy minden egyes literálnak (x_i vagy $\bar{x}_i$) megfeleltetünk z különböző elemet, így $t = 2mz$ és

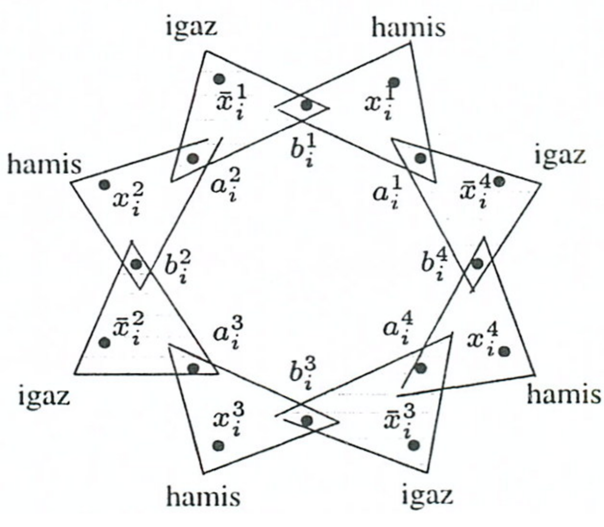
$$U_1 = \{x_i^j, \bar{x}_i{}^j \mid 1 \leq i \leq m, 1 \leq j \leq z\}.$$

Az x_i változó helyettesítéseinek a következő hármasokat feleltetjük meg:

$$E_i^{igaz} = \{(\bar{x}_i{}^j, a_i^j, b_i^j) \mid 1 \leq j \leq z\},$$

$$E_i^{hamis} = \{(x_i^j, a_i^{j+1}, b_i^j) \mid 1 \leq j \leq z\},$$

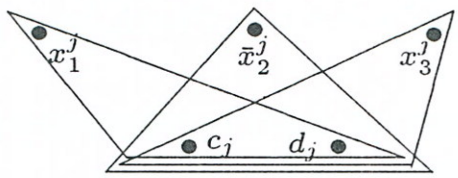
ahol a j szerinti indexelést ciklikusan végezzük, azaz $a_i^{z+1} = a_i^1$. Könnyen látható, hogy az $\{a_i^j \mid j = 1, \ldots, z\} \cup \{b_i^j \mid j = 1, \ldots, z\}$ halmaz $E_i^{igaz} \cup E_i^{hamis}$-beli hármasokkal csak úgy fedhető le egyrétűen egy rögzített i-re, hogy vagy az E_i^{igaz}, vagy az E_i^{hamis} rendszert vesszük S'-be. Előbbi esetben az U_1 halmazból a z darab $x_i = igaz$ értéknek megfelelő $x_i^1, \ldots, x_i^z$ elem, utóbbi esetben pedig a z darab $x_i = hamis$ értéknek megfelelő $\bar{x}_i{}^1, \ldots, \bar{x}_i{}^z$ elem marad lefedetlenül.



A ϕ_j részformulának feleltessük meg a következő hármasokat:

$$F_j = \{(x_i^j, c_j, d_j) \mid x_i \text{ literálja } \phi_j\text{-nek}\} \cup \{(\bar{x}_i^j, c_j, d_j) \mid \bar{x}_i \text{ literálja } \phi_j\text{-nek}\}.$$

Az F_j halmaznak legfeljebb 3 eleme van, mert a ϕ_j legfeljebb 3 literált tartalmaz. Például a $\phi_j = x_1 \vee \overline{x}_2 \vee x_3$ részformulának megfelelő hármasok így szemléltethetők:



Rögzített j-re a c_j, valamint a d_j elemek egyrétű lefedése az F_j halmaz egyetlen elemének a kijelölését jelenti. Az eddigiek alapján az alábbi H halmaz

$$\{a_i^j \mid i = 1, \ldots, m,\ j = 1, \ldots, z\} \cup \{c_j \mid j = 1, \ldots, z\} \cup$$
$$\cup \{b_i^j \mid i = 1, \ldots, m\ ; j = 1, \ldots, z\} \cup \{d_j \mid j = 1, \ldots, z\}$$

pontosan akkor fedhető le egyrétűen $\bigcup_{i=1}^{m}(E_i^{igaz} \cup E_i^{hamis}) \cup \bigcup_{j=1}^{z} F_j$-beli hármasokkal, ha a $\phi_1, \ldots, \phi_z$ formulák együttesen kielégíthetők. A c_j-t fedő hármas felel meg egy *igaz* értékű literálnak ϕ_j-ből, az a_i^j elemek pedig biztosítják, hogy ez a kijelölés megfelel a változók egy kiértékelésének (ha valahol az u literált választjuk igaznak, akkor a $\bar{u}$ literált sehol sem választhatjuk igaznak).

Már csak az van hátra, hogy kiegészítsük az eddigi konstrukciót a 3-DH egy inputjává. Az a_i^j, valamint a c_j elemeket az U_2 halmaz elemeinek tekintjük, a b_i^j, valamit a d_j elemeket pedig az U_3 halmaz elemeinek. Ki kell egészíteni az U_2 és U_3 halmazokat $t = 2mz$ eleművé. Az eddigi hármasok rendszerét pedig úgy kell bővíteni, hogy a H halmaz fenti szerkezetű egyrétű fedései, és csak azok legyenek kiegészíthetők az $U_1 \cup U_2 \cup U_3$ halmaz egyrétű lefedésévé. Ezek nyilvánvalóan teljesülnek, ha beveszünk U_2-be, valamint U_3-ba még egyenként $z(m-1)$ elemet: $e_1, \ldots, e_{z(m-1)}$-et, illetve $f_1, \ldots, f_{z(m-1)}$-et, továbbá a hármasok rendszerét kiegészítjük a

$$G = \left\{ (x_i^j, e_k, f_k), (\bar{x}_i{}^j, e_k, f_k) \;\middle|\; \begin{array}{l} i = 1, \ldots, m, \\ j = 1, \ldots, z, \\ k = 1, \ldots, z(m-1) \end{array} \right\}$$

halmazzal. Beláttuk tehát, hogy az eredeti $\phi \in$ 3-CNF pontosan akkor kielégíthető, ha az

$$\begin{aligned}
U_1 &= \{x_i^j, \bar{x}_i{}^j \mid i = 1, \ldots, m,\ j = 1, \ldots, z\}, \\
U_2 &= \{a_i^j, c_j, e_k \mid i = 1, \ldots, m,\ j = 1, \ldots, z,\ k = 1, \ldots, z(m-1)\}, \\
U_3 &= \{b_i^j, d_j, f_k \mid i = 1, \ldots, m,\ j = 1, \ldots, z,\ k = 1, \ldots, z(m-1)\}, \\
S &= G \cup \bigcup_{i=1}^{m} (E_i^{igaz} \cup E_i^{hamis}) \cup \bigcup_{j=1}^{z} F_j
\end{aligned}$$

inputra a 3-DH feladat megoldható. A konstrukció polinom időben elvégezhető; a tételt igazoltuk.□

Egy szorosan kapcsolódó érdekes probléma a *Pontos fedés hármasokkal*: adott egy U véges halmaz, és U háromelemű részhalmazainak egy $\mathcal{F} = \{X_1, X_2, \ldots, X_k\}$ családja. Eldöntendő, hogy az $\mathcal{F}$-ből kiválaszthatók-e páronként diszjunkt halmazok, melyek együttesen lefedik U-t. Jelölje X3C azokat az $(U, \mathcal{F})$ párokat, melyekre az $\mathcal{F}$-ből kiválasztható U ilyen értelemben vett pontos fedése.

Példa: Legyen $U = \{1, 2, 3, \ldots, 9\}$,

$$\begin{aligned}
\mathcal{F} &= \{\{1,2,3\}, \{1,4,5\}, \{1,6,7\}, \{1,8,9\}\} \text{ és} \\
\mathcal{F}' &= \{\{1,2,3\}, \{4,5,6\}, \{1,6,7\}, \{7,8,9\}\}.
\end{aligned}$$

Ekkor $\mathcal{F}$-ből nem választható ki pontos fedés, hiszen az 1 benne van mindegyik hármasban. Tehát $(U, \mathcal{F}) \notin$ X3C. A $\mathcal{F}'$ viszont tartalmazza U egy pontos fedését, ezért $(U, \mathcal{F}') \in$ X3C.

Állítás: *Az X3C nyelv* NP-*teljes.*

Bizonyítás: X3C $\in$ NP teljesül; tanú lehet egy pontos fedés. Megmutatjuk ezután, hogy 3-DH $\prec$ X3C. Legyen $U_1, U_2, U_3; S$ a 3-DH egy inputja. Az elemek esetleges átnevezése után feltehetjük, hogy az U_i halmazok páronként diszjunktak. Legyen $U = U_1 \cup U_2 \cup U_3$, és $\mathcal{F} = \{\{a, b, c\} | \ (a, b, c) \in S\}$. Könnyű meggondolni, hogy az így kapott megfeleltetés Karp-redukció. Ennek a részleteit az olvasóra bízzuk. □

8.7.5. Hamilton-kört tartalmazó gráfok és az Utazó ügynök probléma

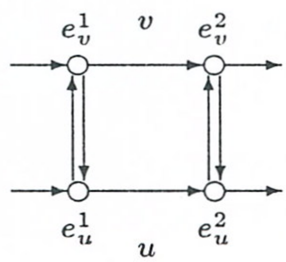
Korábban már foglalkoztunk az irányítatlan, illetve irányított Hamilton-kört tartalmazó gráfokkal, megállapítva, hogy a H és IH nyelvek NP-beliek. Most megmutatjuk, hogy ezek is nehéz nyelvek.

Tétel: *Az IH nyelv* NP*-teljes.*

Bizonyítás: Megmutatjuk, hogy *Éllefogás* $\prec$ IH. Legyen $G = (V, E)$ egy irányítatlan gráf, k pedig egy pozitív egész. Feltehetjük, hogy $k \leq |V(G)|$. A (G, k) pár segítségével hatékonyan (polinom időben) elkészítünk egy $G' = (V', E')$ irányított gráfot, melyben pontosan akkor lesz irányított Hamilton-kör, ha G élei lefoghatók k csúccsal. A G minden $e = (u, v) \in E$ éléhez elkészítjük a

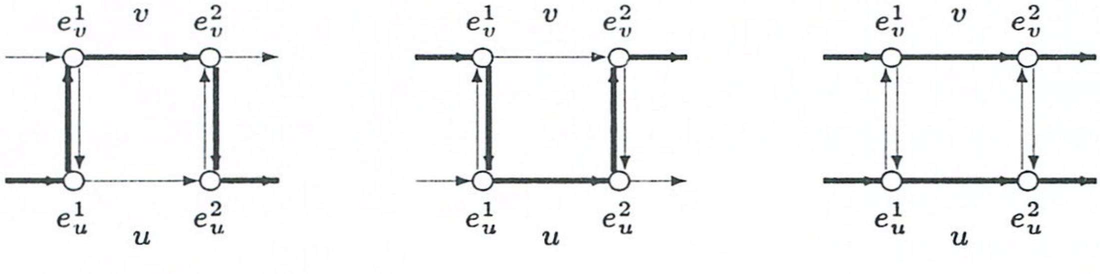
$$\begin{aligned} G'_e &= (V'_e, E'_e) = \{e^1_u, e^2_u, e^1_v, e^2_v\}, \\ &\quad \{e^1_u \to e^2_u,\ e^1_v \to e^2_v,\ e^1_u \to e^1_v,\ e^1_v \to e^1_u,\ e^2_u \to e^2_v,\ e^2_v \to e^2_u)\}) \end{aligned}$$

irányított gráfot.



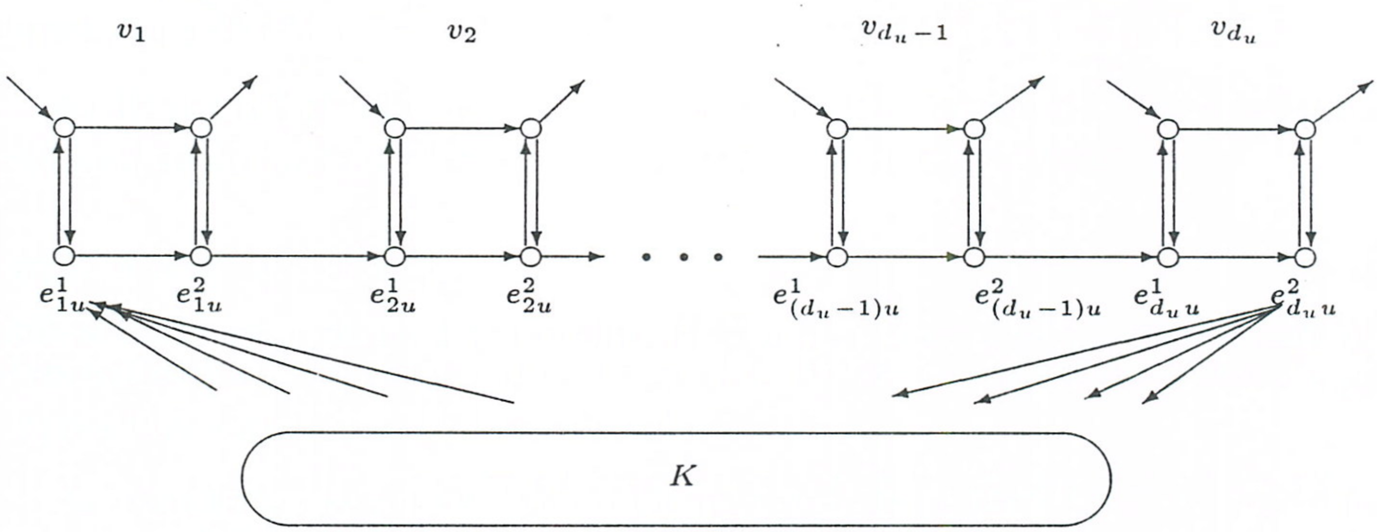
A G' gráfot jórészt ilyen G'_e alakú kis részgráfokból fogjuk felépíteni. Egy G'_e gráfba kívülről csak az e^1_u, illetve az e^1_v csúcsokba futnak élek, kifelé pedig csak az e^2_u, illetve az e^2_v csúcsokból mennek élek. Ezekkel a megkötésekkel könnyű belátni, hogy a felépítendő G' gráf egy Hamilton-köre csak a következő háromféle módon haladhat át a $G'_{u,v}$ részgráfon:

(1) $\cdots \to e^1_u \to e^1_v \to e^2_v \to e^2_u \to \cdots$
(2) $\cdots \to e^1_v \to e^1_u \to e^2_u \to e^2_v \to \cdots$
(3) $\cdots \to e^1_u \to e^2_u \to \cdots \to e^1_v \to e^2_v \to \cdots$



Ezek után a G' gráf felépítése a következő: Vegyük G minden egyes e éléhez a G'_e gráfot. A G minden $u \in V$ csúcsának megfelelően fűzzük fel egy irányított útra tetszőleges sorrendben (pl. az éllista szerinti sorrendben) az u-hoz csatlakozó

éleknek megfelelő G'_e részgráfoknak az u csúcshoz tartozó felét ($\{e^1_u, e^2_u\}$). Vegyünk még egy k elemű K ponthalmazt, és K összes eleméből vezessünk élet az előbb megadott utak kezdőpontjához, továbbá vezessünk K minden pontjába élet az utak végpontjaiból. Az $u \in V$ csúcshoz tartozó irányított út így szemléltethető (az u foka d_u):



A formális leírások kedvelői számára legyen E'_u a következő élhalmaz:

$$E'_u = \{w \to e^1_{1u},\ e^2_{1u} \to e^1_{2u},\ \dots,\ e^2_{(d_u-1)u} \to e^1_{d_u u},\ e^2_{d_u u} \to w \mid w \in K\},$$

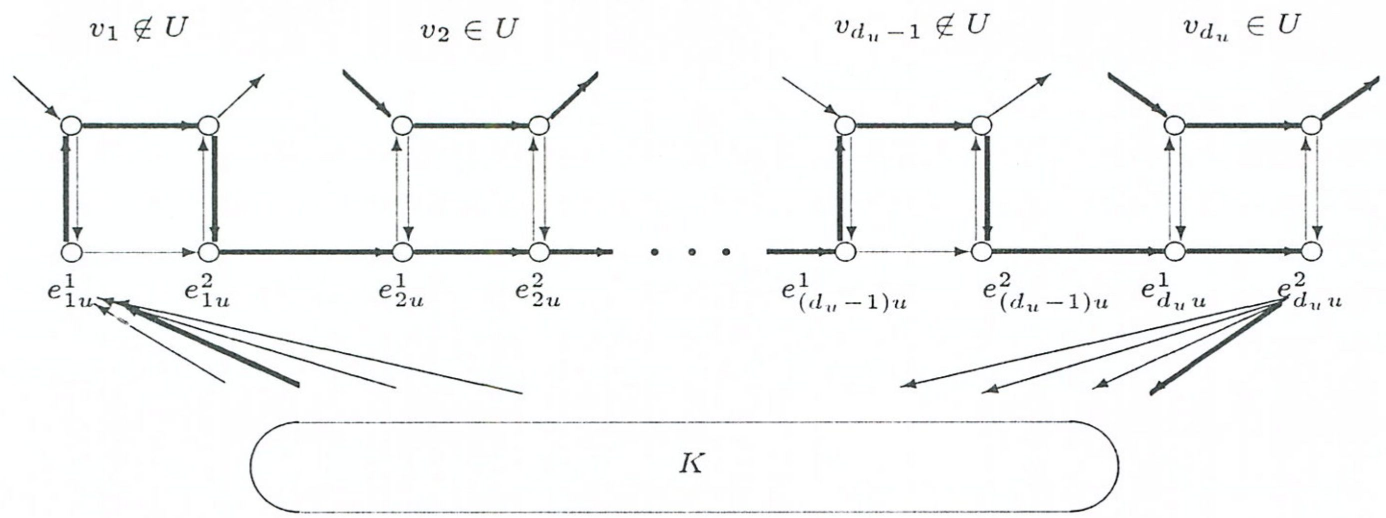
ahol $e_1, \dots, e_{d_u}$ az u-hoz csatlakozó G-beli élek listája; ezután G' így adható meg:

$$V' = K \cup \bigcup_{e \in E} V'_e, \quad E' = \bigcup_{e \in E} E'_e \cup \bigcup_{u \in V} E'_u$$

Tegyük fel most, hogy G' tartalmaz egy H irányított Hamilton-kört. A kis gráfokkal kapcsolatos (1)-(3) bejárási lehetőségeket figyelembe véve állíthatjuk, hogy ha valamely $u \in V$ csúcs esetén H tartalmaz legalább egy élet az E'_u halmazból, akkor ezen él a H egy ilyen szerkezetű darabján van:

$$\dots \to w_j \to e^1_{1u} \rightsquigarrow e^2_{1u} \to e^1_{2u} \rightsquigarrow e^2_{2u} \to \dots \to e^1_{d_u u} \rightsquigarrow e^2_{d_u u} \to w_{j+1} \to \dots .$$

Itt $w_j, w_{j+1} \in K$, és egy hullámos nyíl vagy egy (1) típusú 3 hosszú utat jelöl a G'_{e_i} részgráfon belül vagy pedig az $e^1_{iu} \to e^2_{iu}$ élet (utóbbi esetben szükségképpen a H egy másik darabja tér vissza G'_{e_i} maradék két csúcsát meglátogatni).



A K halmaznak k eleme van, ezért a H Hamilton-kör k darab ilyen útból tevődik össze:

$$\begin{aligned} w_1 \to u_1\text{-nek megfelelő rész} &\to w_2 \to u_2 \to \cdots \\ \cdots &\to w_k \to u_k\text{-nak megfelelő rész} \to w_1 . \end{aligned}$$

A H mindegyik kis G'_e ($e \in E$) csúcsait bejárja. Ez azt jelenti, hogy az $u_1, \ldots, u_k$ csúcsok együttesen lefogják G éleit. Van tehát G-nek k elemű éllefogó ponthalmaza.

Fordítva, tegyük fel, hogy $U = \{u_1, \ldots, u_k\}$ egy éllefogó pontrendszer G-ben. Ekkor a következő utak összefűzve egy Hamilton-kört adnak a G' gráfban:

$$\begin{aligned} \cdots \to w_j \to e^1_{1u_j} \rightsquigarrow e^2_{1u_j} \to e^1_{2u_j} &\rightsquigarrow e^2_{2u_j} \to \cdots \\ \cdots \to e^1_{d_{u_j}u_j} &\rightsquigarrow e^2_{d_{u_j}u_j} \to w_{j+1} \to \cdots , \end{aligned}$$

ahol $w_{k+1} = w_1$; a hullámos nyilak G'_e-beli utakat jelölnek, mégpedig ha e-nek a másik végpontja nincs U-ban, akkor a 3 hosszú utat, ha pedig benne van, akkor az 1 hosszút. Utóbbi esetben az e másik végpontjának megfelelő részben található meg az út párja.

Beláttuk tehát, hogy G-ben pontosan akkor van k elemű éllefogó pontrendszer, ha G'-ben van irányított Hamilton-kör. A G' gráf konstrukciója hatékonyan elvégezhető. Ezzel a bizonyítást befejeztük. □

A DAG-ok algoritmusainak tárgyalásakor (6.4.2. rész) foglalkoztunk a két pont közötti leghosszabb egyszerű utak meghatározásának a feladatával. Kiderült, hogy ez villámgyorsan, vagyis lineáris uniform költséggel megoldható, amikor a bemeneti G gráf egy DAG. Az előző tétel masszív érvet szolgáltat amellett, hogy a feladat nehézzé válik, ha a G tetszőleges irányított gráfot jelenthet. Ha volna

ugyanis egy gyors (mondjuk polinom idejű) módszerünk, amely megmondaná a G gráf u, v pontjaira a leghosszabb egyszerű $u \rightsquigarrow v$ irányított utak $l(u, v)$ hosszát, akkor az IH nyelvet is gyorsan el tudnánk dönteni.

Feladat: Igazoljuk az előző mondatban foglalt állítást. (Legyen a $G = (V, E)$ irányított gráf az IH egy bemenete. Rendeljünk egységnyi súlyokat az éleihez. Mutassuk meg, hogy a G egy $u \to v$ éle pontosan akkor éle egy irányított Hamilton-körnek, ha a G-ből az $u \to v$ törlésével kapott gráfban $l(v, u) = |V| - 1$.)

Az IH nyelv NP-teljességét tudva az irányítatlan Hamilton-köröket tartalmazó gráfokról szóló hasonló állítással elég egyszerűen elboldogulunk.

Tétel: *A H nyelv* NP-*teljes.*

Bizonyítás: Korábban már láttuk, hogy H $\in$ NP, és az első Karp-redukció, amit megnéztünk, egy IH $\prec$ H leképezés volt. □

Az irányítatlan Hamilton-kör probléma nevezetes általánosítása az *Utazó ügynök* feladat: adott egy G irányítatlan gráf pozitív egészekkel súlyozott élekkel. A cél minél rövidebb összsúlyú Hamilton-kört találni G-ben. (Az ügynök szeretné minél kisebb költséggel sorba látogatni az adott városokat, mindegyiket csak egyszer, napszálltával hazatérve.) A feladathoz kézenfekvően kapcsolódó Ut nyelv olyan (G, k) párokból áll, melyekben G egy súlyozott élű irányítatlan gráf, k egy nemnegatív egész, és G-ben van k-nál nem nagyobb súlyú Hamilton-kör. Világos, hogy $Ut \in$ NP. A H feladat az Ut feladat egy speciális esetének tekinthető: minden élsúly 1 és $k = |V(G)|$. Az a leképezés tehát, amely a G gráfhoz a csupa egyessel súlyozott élű G-t és a $k = |V(G)|$ korlátot rendeli, egy H $\prec Ut$ Karp-redukciót ad meg. Érvényes a következő:

Tétel: *Az Ut nyelv* NP-*teljes.* □

8.7.6. A Hátizsák feladat és néhány más rokon probléma

Itt olyan kérdésekről lesz szó, melyekben az eddigiekhez képest több szerep jut a számoknak. Az első a *Hátizsák* feladat: adottak az $s_1, \ldots, s_m > 0$ súlyok, ezek $v_1, \ldots, v_m > 0$ értékei, valamint a b megengedett maximális összsúly. Tegyük fel, hogy az s_i, v_i, b számok egészek. A feladat az, hogy találjunk egy olyan $I \subseteq \{1, .., m\}$ részhalmazt, melyre $\sum_{i \in I} s_i \leq b$, és ugyanakkor $\sum_{i \in I} v_i$ a lehető legnagyobb. Szemléletes képként gondoljunk arra, hogy van egy b teherbírású hátizsákunk. Ebbe akarunk belerakni bizonyosakat az s_i súlyú tárgyak közül úgy, hogy a hátizsákban levő dolgok összértéke a lehető legnagyobb legyen. Másfelől

a hátizsákot kímélni kell: a bepakolt tárgyak összsúlya nem lehet több b-nél. Az I indexhalmaz mondja meg, hogy a súlyok közül melyeket raktuk a hátizsákba.

Polinom idejű extra munka erejéig azonos nehézségű NP-beli feladatot kapunk, ha bevezetünk még egy $k > 0$ értékkorlátot és egy adott $s_1, \ldots, s_m$; $v_1 \ldots, v_m; b; k$ inputra azt kérdezzük, hogy van-e olyan kímélő kitöltése a hátizsáknak, melynek az értéke legalább k. Az így kapott döntési feladat (nyelv) neve legyen *Hát*.

$$\textit{Hát} = \{(s_1, s_2, \ldots, s_m; v_1, v_2, \ldots, v_m; b; k) | s_i, b, k > 0 \text{ egészek, és}$$
$$\text{van olyan } I \subseteq \{1, \ldots, m\} \text{ melyre } \sum_{i \in I} s_i \leq b \text{ és } \sum_{i \in I} v_i \geq k\}.$$

Feladat: Mutassuk meg, hogy *Hát* $\in$ P-ből *Hátizsák* $\in$ FP következne. (A prímtényezős felbontás és az F feladat viszonyát tisztázó állítás ötlete használható: bináris kereséssel meghatározhatjuk a legnagyobb elérhető k-t.)

Igazából úgy gondoljuk, hogy a valóságban nem teljesül a feladat feltétele, a *Hát* nyelv nem ismerhető fel polinom időben. Ezt alátámasztandó megmutatjuk, hogy a *Hát* nyelv NP-teljes. Egészen pontosan azt a speciális esetet fogjuk szemügyre venni, amikor a tárgyak súlya és értéke azonos, valamint a súlykorlát megegyezik az értékkorláttal: $s_i = v_i$ $(i = 1, 2, \ldots, m)$ és $b = k$. Ez *Részhalmazösszeg probléma*. Itt arról az egyszerűbb algoritmikus kérdésről van szó, hogy pontosan tele lehet-e rakni a hátizsákot az adott tárgyakból válogatva. A megfelelő nyelv

$$\mathrm{RH} = \left\{ (s_1, \ldots, s_m; b) \,\middle|\, \begin{array}{l} s_i, b > 0 \text{ egészek,} \\ \text{és van olyan } I \subseteq \{1, \ldots, m\} \text{ hogy } \sum_{i \in I} s_i = b \end{array} \right\}.$$

Tétel: *Az RH nyelv* NP-*teljes.*

Bizonyítás: Világos, hogy RH $\in$ NP; alkalmas I indexhalmaz lehet a tanú. Ezután megadunk egy X3C $\prec$ RH Karp-redukciót. Vegyük X3C tetszőleges inputját: ez egy U véges halmaz és egy $\mathcal{F} = \{X_1, \ldots, X_m\}$ halmazrendszer, melyre $X_i \subseteq U$ és $|X_i| = 3$. Ebből az RH feladat egy inputját fogjuk hatékonyan elkészíteni. Tegyük fel, hogy $U = \{1, 2, \ldots, t\}$, és legyenek

$$\begin{aligned}
r &= 1 + \binom{t}{2}, \\
z_j &= r^{j-1} \quad (j = 1, \ldots, t), \\
s_i &= \sum_{j \in X_i} z_j \quad (i = 1, \ldots, m), \\
b &= \sum_{j=1}^{t} z_j.
\end{aligned}$$

Lényegében az történik, hogy az X_i halmazokat elég nagy (itt konkrétan r) alapú számrendszerben felírt számokkal kódoljuk. Az $x = (s_1, \dots, s_m; b)$ együttest fogjuk az RH inputjaként értelmezni. Az x hossza $O(mt \log t)$, mert $\log s_i = O(\log z_t) = O(t \log t)$, ezért x a megadott formulák szerint (t-ben) polinom időben előállítható. A megfeleltetés tehát teljesíti a Karp-redukcióval szemben támasztott hatékonysági követelményt.

Nézzük ezután a másik előírást. Legyen $I \subseteq \{1, \dots, m\}$ egy indexhalmaz. Az $f_j = |\{i | i \in I \text{ és } j \in X_i\}|$ számokat az I-hez tartozó *fedési számoknak* nevezzük ($j = 1, \dots, t$). Az f_j fedési szám azt mondja meg, hogy az $i \in I$ indexekhez tartozó X_i halmazok közül hány tartalmazza a $j \in U$ elemet. Ezután megmutatjuk, hogy az I-hez tartozó $\sum_{i \in I} s_i$ részhalmazösszeg és az I fedési számainak $f_1, f_2, \dots, f_t$ sorozata kölcsönösen egyértelműen meghatározzák egymást. A részhalmazösszeg így írható:

$$\sum_{i \in I} s_i = \sum_{i \in I} \sum_{j \in X_i} z_j = \sum_{j=1}^{t} |\{i | i \in I \text{ és } j \in X_i\}| r^{j-1} = \sum_{j=1}^{t} f_j r^{j-1}.$$

A $j \in U$ pontot legfeljebb $\binom{t}{2}$ számú hármas tartalmazza, ezért a jobb oldalon $\binom{t}{2} < r$ miatt r^{j-1} együtthatója kisebb, mint r. Eszerint az f_j fedési számok éppen az I-nek megfelelő részhalmazösszeg r alapú számrendszerbeli jegyei. Az r alapú számrendszerbeli felírás egyértelműsége miatt a b szám tehát akkor és csak akkor lesz az I-nek megfelelő részhalmazösszeg, ha az összes fedési szám egy, ami éppen annyit tesz, hogy az $\{X_i | i \in I\}$ halmazrendszer pontosan fedi U-t. Tehát b akkor és csak akkor részhalmazösszeg, ha létezik pontos fedés. □

A Részhalmazösszeg probléma egy érdekes speciális esete a *Partíció* feladat, ahol a b megcélzott összeg az s_i számok összegének a fele:

$$\textit{Partíció} = \left\{ (s_1, \dots, s_m) \;\middle|\; \begin{array}{l} s_i > 0 \text{ egészek, és van olyan} \\ I \subseteq \{1, \dots, m\}, \text{ hogy } \sum_{i \in I} s_i = \frac{1}{2} \sum_{i=1}^{m} s_i \end{array} \right\}.$$

Arról a kérdésről van szó, hogy az $\{1, 2, \dots, m\}$ halmaz felbontható-e két egyenlő részhalmazösszeget adó részre. Nyilvánvaló, hogy *Partíció* $\in$ NP.

Feladat: Mutassuk meg, hogy a *Partíció* nyelv NP-teljes, megadva egy RH $\prec$*Partíció* redukciót. (Vegyük az RH egy $x = (s_1, \dots, s_m; b)$ inputját. Feltehető, hogy $b \leq s = \sum_{i=1}^{m} s_i$. Az x-hez rendeljük a *Partíció* következő inputját: $(s_1, \dots, s_m, s + 1 - b, b + 1)$. Itt a számok összege $2s + 2$. Az utolsó két szám nem lehet egy partíció ugyanazon osztályában, mert az összegük túl nagy: $s + 2$.)

A *Ládapakolás* feladat a következő: adottak az $s_1, \dots, s_m$ súlyú tárgyak, $0 < s_i \leq 1$, s_i racionális, és egy $k > 0$ egész. Eldöntendő, hogy a tárgyakat bele lehet-e pakolni legfeljebb k számú egységnyi súlykapacitású ládába.

Feladat: Mutassuk meg, hogy *Ládapakolás* feladat nyelve NP-teljes. (Adjunk meg egy *Partíció*$\prec$*Ládapakolás* Karp-redukciót; elegendő két ládára szorítkozni, azaz a $k = 2$ eset is megteszi.)

A *Ládapakolás* problémával a következő fejezetben is találkozunk. A feladat (közelítő) megoldására szolgáló algoritmusokat fogunk ismertetni.

8.7.7. Lineáris programozás

Itt egy igen széles körben előforduló feladatról lesz szó. Alkalmazásaival találkozhatunk egyebek között a gazdasági számítások és a mérnöki tervezés területén. Egyszerűen fogalmazva, változók lineáris függvényét kell optimalizálni (maximalizálni, vagy minimalizálni) egy olyan tartományban, amelyet a változókra kirótt lineáris egyenlőtlenségek írnak le. Kezdjük egy nevezetes, gazdasági tónusú példával:

Példa: *Termékszerkezet-feladat*
Egy üzem n-féle terméket tud termelni; ennek során m-féle erőforrást használ fel. Az erőforrások jelenthetnek pl. munkaórákat, anyagokat, gépidőket, stb. Legyen c_j a j-edik termék egységnyi mennyiségére eső nyereség, b_i az i-edik erőforrásból rendelkezésre álló mennyiség, továbbá a_{ij} az i-edik erőforrásból az a mennyiség, ami a j-edik termék egy egységének előállításához szükséges. Az üzem vezetésének a célja egy olyan termékszerkezet kialakítása, ami az előbb leírt körülmények között maximalizálja a nyereséget.

Jelölje x_j a j-edik termékből előállítani kívánt mennyiséget. A feladat így fogalmazható:

$$(*)\qquad \text{maximalizáljuk a} \quad \sum_{j=1}^{n} c_j x_j \text{mennyiséget,}$$

$$\text{feltéve, hogy} \quad \sum_{j=1}^{n} a_{ij} x_j \leq b_i, \quad i = 1, 2, \ldots, m,$$

$$\text{továbbá} \quad x_j \geq 0, \quad j = 1, 2, \ldots, n.$$

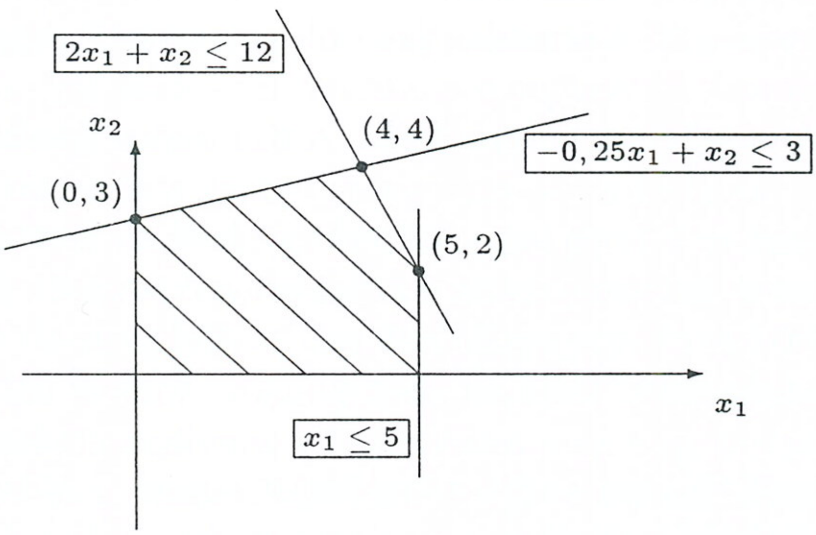
A $(*)$ alakban felírható feladatokat – ahol a c_j, b_i és a_{ij} tetszőleges racionális számok –, *lineáris programozási feladat*oknak nevezzük. A feladat *megoldásán* egy olyan racionális számokból álló $(x_1, x_2, \ldots, x_n)$ vektort értünk, melyre a $\sum_{j=1}^{n} c_j x_j$ mennyiség (az ún. *célfüggvény* értéke) maximális az előírt egyenlőtlenségeket teljesítő vektorok között.

Egy lineáris programozási feladatot tehát az n, m paraméterekkel, a célfüggvényt jelentő $(c_1, \ldots, c_n)$ vektorral, valamint a feltételeket leíró $A = [a_{ij}]$ $(1 \leq i \leq m, 1 \leq j \leq n)$ mátrixszal és $(b_1, \ldots, b_m)$ vektorral adhatunk meg.

Példa: Legyen $n = 2$, $m = 3$, $c_1 = c_2 = 1$,

$$A = \begin{pmatrix} 1 & 0 \\ 2 & 1 \\ -0{,}25 & 1 \end{pmatrix},$$

továbbá $b_1 = 5$, $b_2 = 12$ és $b_3 = 3$. Az $x_1 + x_2$ mennyiség maximumát keressük az $x_1 \leq 5$, $2x_1 + x_2 \leq 12$, $-0,25x_1 + x_2 \leq 3$, $x_1 \geq 0$ és $x_2 \geq 0$ egyenlőtlenségek által meghatározott tartományon. Ez egy sokszöglemez az (x_1, x_2)-síkon:



A második egyenlőtlenség ötszöröséhez adjuk hozzá a harmadik négyszeresét: azt kapjuk, hogy $9x_1 + 9x_2 \leq 72$. Innen már látszik, hogy a célfüggvény optimuma legfeljebb 8; ezt a $(4, 4)$-ben el is éri. A $(4, 4)$ pont az egyik csúcsa a feltételeket teljesítő pontokból álló alakzatnak.

Sokáig nyitott kérdés volt, hogy a lineáris programozási feladat mint algoritmikus probléma milyen nehéz. A megoldására kidolgozott eljárások közül a legnépszerűbb és leghasznosabb az ún. *szimplex-módszer*. Ennek ismertetése itt nem célunk. A módszert George Dantzig javasolta a negyvenes évek végén, és mostanra szinte művészi tökéllyel megírt, rendkívül csiszolt implementációi születtek[2]. Úgy tűnik, a gyakorlatilag fontos feladatok széles körében igen gyors: $O(m^2n)$ aritmetikai művelet elegendő a „praktikus" feladatok megoldására. Ezzel szemben ismertek olyan példák, amelyeken a módszer exponenciális nagyságrendű műveletet végez. A szimplex-módszer a feltételek által adott alakzat csúcsaiban keresi

[2]Az egyik legnépszerűbb ezek közül a MINOS nevű rendszer.

az optimumot[3]. Megadhatók olyan $2d$ feltételt és d változót tartalmazó ($m = 2d$, $n = d$) lineáris programozási feladatok, amelyeknél a szimplex-módszer egyik legjobb megfigyelt viselkedésű változata $2^d - 1$ csúcsot vizsgál meg.

Nagy visszhangot kiváltó áttörést jelentett L. G. Hacsijan 1979-ben javasolt eljárása, az *ellipszoid-módszer*. Az ellipszoid-módszer az első polinom idejű algoritmus a lineáris programozási feladatok megoldására. Hacsijan eredménye úgy is fogalmazható, hogy a probléma az FP osztályban van. Az algoritmus a gyakorlati feladatok körében eddig nem tudott versenyezni a szimplex-módszerrel. Jelentősége ezért elsősorban elméleti. Ilyen értelemben viszont erős eszköznek számít. Több érdekes kombinatorikus optimalizálási feladatra az ellipszoid-módszer segítségével sikerült polinom idejű algoritmust nyerni.

1984-ben Narendra K. Karmarkar javasolt egy az addigiaktól merőben eltérő szemléletű algoritmust, a *belső pont módszert*. Ez – csakúgy, mint Hacsijan módszere – szintén polinomkorlátos algoritmus. A Karmarkar gondolatai nyomán született eljárások a gyakorlatban is versenyezni tudnak a szimplex-módszerrel. Több esettanulmány mutat arra, hogy nagy méretű (gyakorlati) feladatok megoldására a belső pont módszerek hatékonyabbak a szimplex-módszernél. Egyes szerzők szerint $m + n > 2500$ esetén az új technikák már sokkal gyorsabbak.

A lineáris programozás feladata tehát polinom időben megoldható. Hasonló mondható a megfelelő igen-nem feladatról. Ez annyiban különbözik $(*)$-tól, hogy a c_j, b_i és a_{ij} adatok mellett még egy racionális szám, a K korlát is része a bemenetnek. Az eldöntendő kérdés pedig az, hogy van-e olyan racionális számokból álló $(x_1, x_2, \ldots, x_n)$ vektor, melyre a $(*)$-beli egyenlőtlenségek teljesülnek, és $\sum_{j=1}^{n} c_j x_j \geq K$.

Egész értékű lineáris programozás

A lineáris programozási probléma „diszkrét" változata az *egész értékű lineáris programozási feladat*. Ezen olyan, a $(*)$ alakban felírható feladatot értünk, amelyben a c_j, b_i és a_{ij} adatok egészek, és az optimumot az egész koordinátájú $(x_1, x_2, \ldots, x_n) \in \mathbb{Z}^n$ vektorok körében keressük. A megfelelő nyelv (igen-nem probléma) a racionális esethez hasonlóan kapható meg. Álljon az EP nyelv azokból a $(K;\; c_1, \ldots, c_n;\; b_1, \ldots, b_m;\; a_{11}, a_{12}, \ldots, a_{mn})$ bemenetekből

[3]Az előző síkbeli példához hasonlóan magasabb dimenzióban ($n > 2$) is beszélhetünk az egyenlőtlenségek által kijelölt tartomány csúcsairól. Ezek az olyan $(x_1, \ldots, x_n)$ pontok közül kerülnek ki, amelyekre az $n + m$ egyenlőtlenség közül n-ben egyenlőség igaz. Megmutatható, hogy ha a feladatnak van egyáltalán megoldása (véges optimuma), akkor a csúcspontok egyike is megoldás lesz.

$(K, c_j, b_i, a_{ij} \in \mathbb{Z})$, amelyekhez van olyan $(x_1, x_2, \ldots, x_n) \in \mathbb{Z}^n$ vektor, hogy

$$\sum_{j=1}^{n} c_j x_j \geq K, \quad \sum_{j=1}^{n} a_{ij} x_j \leq b_i, \quad i = 1, 2, \ldots, m, \quad \text{és } x_j \geq 0, \quad j = 1, 2, \ldots, n.$$

Megmutatható – ennek a részleteire nem térünk ki –, hogy az EP nyelv az NP osztályban van. Egy megfelelő $(x_1, x_2, \ldots, x_n) \in (\mathbb{Z}^+)^n$ vektor szolgálhat tanúként. Az is igaz, hogy az egész értékű optimalizálási feladat és az EP nyelv felismerése polinomiálisan ekvivalens nehézségűek. Tehát ha létezne polinom idejű algoritmus EP felismerésére, akkor az optimalizálási feladat FP-ben lenne. A következő állítást úgy is értelmezhetjük, hogy ez nem túlságosan valószínű.

Tétel: *Az EP nyelv* NP-*teljes.*

Bizonyítás: Megmutatjuk, hogy RH $\prec$ EP, ahol RH a Részhalmazösszeg feladat nyelve. Az RH egy $(s_1, \ldots, s_n; b)$ bemenetéhez rendeljük a következő egyenlőtlenségeket: $x_j \leq 1$, $j = 1, 2, \ldots, n$, és $\sum_{j=1}^{n} s_j x_j \leq b$. A célfüggvény legyen $\sum_{j=1}^{n} s_j x_j$, a korlát pedig $K = b$. Ezzel az EP egy bemenetét kaptuk; n változónk és $n + 1$ feltételünk van.

Az így szerkesztett bemenet pontosan akkor van EP-ben, ha létezik olyan $(x_1, x_2, \ldots, x_n) \in (\mathbb{Z}^+)^n$ vektor, amelyre teljesülnek a feltételi egyenlőtlenségek, és $\sum_{j=1}^{n} s_j x_j \geq b$. Az utolsó feltétel miatt ekkor $\sum_{j=1}^{n} s_j x_j = b$. Az x_j számok nemnegatív egészek, így az értékük csak 0 vagy 1 lehet. Legyen $I = \{j;\ x_j = 1\}$. Az $\{s_j;\ j \in I\}$ részhalmaz összege éppen b.

Fordítva, ha az $\{s_j;\ j \in I\}$ részhalmaz összege b, akkor az

$$x_j = 0, \text{ ha } j \notin I \text{ és } x_j = 1, \text{ ha } j \in I$$

feltételekkel megadott vektor teljesíti az egyenlőtlenségeket. A megfeleltetés tényleg Karp-redukció. □

Az egész értékű lineáris programozás két szempontból fontos. Egyfelől rendkívül sok érdekes optimalizálási feladat fogalmazható meg ebben a keretben. Az alkalmazási területeknek se szeri, se száma (pl. termékterítés, gyártásszervezés, VLSI-tervezés, kommunikációs és szállítási hálózatok tervezése). Ennek az erős modellezési képességnek a szemléltetésére ajánljuk az olvasónak, hogy fogalmazzon meg néhány további NP-teljes feladatot egész programozási kérdésként:

Feladat: Adjunk meg (minél egyszerűbb) $L \prec$ EP Karp-redukciókat, ahol L a SAT, a 3-DH, a 3-SZÍN, vagy a *Ládapakolás* nyelvet jelenti.

A másik tényező, ami miatt az egész értékű programozás megkülönböztetett figyelmet érdemel, hogy ezzel a feladattípussal sokat foglalkoztak és foglalkoznak a kutatók. Megoldási módszerek gazdag választéka áll rendelkezésre. Ezek a módszerek ugyan nem polinom idejűek – ilyenek nem is remélhetők, hiszen nehéz feladatról van szó –, de számos esetben használhatónak bizonyultak.

8.7.8. Minimális késésszámú ütemezés

Végezetül megemlítünk egyet a sokféle NP-teljes ütemezési feladat közül. A *Minimális késésszámú ütemezés* feladatának bemenete a következő: elvégzendő munkák (véges) D halmaza; minden feladat egy időegységet igényel, és egyszerre csak egy munkával tudunk foglalkozni. Adott még ezen kívül egy $<$ részbenrendezés (irreflexív, tranzitív reláció) D-n. A $d_1 < d_2$ viszony azt jelenti, hogy a d_1 munkát előbb kell elvégezni, mint d_2-t. Minden egyes $d \in D$ munkához adott a pozitív egész $h(d)$ határidő. Végül adott egy $T > 0$ tűréshatár. A T tűréshatár azt jelenti, hogy legfeljebb T olyan feladat lehet, amivel nem készülünk el határidőre.

Az eldöntendő kérdés az, hogy lehet-e a feladatokat úgy ütemezni, hogy legfeljebb T kivétellel minden feladatot határidőre befejezzünk. Pontosabban fogalmazva: van-e olyan $d_1, d_2, \ldots d_k$ sorrendje a D elemeinek, melyre
1. $d_i < d_j$ esetén $i < j$ teljesül; továbbá
2. legfeljebb T számú i index kivételével fennáll a $h(d_i) \leq i$ egyenlőtlenség?

Az 1. feltétel a $<$ részbenrendezés által meghatározott gráf egy topologikus rendezését követeli meg. Ez önmagában könnyen teljesíthető, hiszen a gráf DAG. A 2. feltétel jelenlétével viszont nehéz feladatot kapunk. Megmutatható, hogy az adott tűréshatárral megoldható ütemezési feladatok nyelve NP-teljes. A MAXKLIKK nyelv viszonylag egyszerűen redukálható erre a nyelvre.

9.

Néhány általános algoritmus-tervezési módszer

Minden felfedezett igazságot szabályként használva fel más igazságok felfedezésére ... számos olyan problémát oldottam meg, amelyet azelőtt igen nehéznek tartottam... RENÉ DESCARTES

Problémákat megoldani, algoritmusokat tervezni kreatív folyamat. Ezért nem is várhatunk biztos recepteket, amelyek segítségével mindig elboldogulunk. De nem is vagyunk teljesen fegyvertelenek: vannak olyan ötletek, módszerek, megközelítések, amelyek sok esetben sikeresnek bizonyultak hatékony algoritmusok tervezésében. Ezek a módszerek konkrét feladatok hatékony megoldásaiból kristályosodtak ki, *váltak szabállyá* a descartes-i értelemben. Olyan általános technikák ezek, amelyek feladatok széles körére alkalmazhatók. Már az eddigiek során is találkoztunk néhány ilyen elvvel.

Az **oszd meg és uralkodj** stratégia lényege, hogy a feladatot megpróbáljuk visszavezetni, „felvágni" ugyanezen feladat sokkal kisebb példányaira úgy, hogy ezek megoldásaiból az eredeti probléma megoldása gyorsan összerakható legyen.

Példák:

1. A bináris keresés során egy kérdéssel a keresési tartomány méretét a felére csökkentjük. A hatékony keresőfák alkalmazásakor is ilyesmi történik. Egy összehasonlítással visszavezetjük a feladatot egy jóval kisebb méretű hasonlóra (feleakkorára, vagy még kisebbre, mint pl. a B-fáknál).
2. Az összefésüléses rendezésnél két feleakkora méretű részfeladatra vágjuk az eredetit. Ezek megoldásából gyorsan adódik – összefésüléssel – a globális megoldás.

3. A gyorsrendezés partíciós lépésének a célja a probléma hatékony kettévágása.

Az oszd meg és uralkodj elv különösen akkor tud eredményes lenni, ha a vágás eléggé egyenletes, más szóval a kapott részek *nagyjából egyenlő méretűek*. Jól mutatja ezt a gyorsrendezés, ami akkor bizonyul lassúnak, ha a partíciós lépésekben kapott vágások nem eléggé egyenletesek. Azt is érdemes meggondolni, hogy az 1. és 2. példák módszereinek elemzésében, az igen kedvező korlátok levezetésében fontos szerepet játszott a vágások egyenletessége.

A **mohó módszer** néhány alkalmazásával is találkoztunk már. Az elv valami olyasmit mond, hogy a feladat megoldása során a következő lépés legyen a lehető legjobb a rendelkezésre állók közül; itt a *legjobb* valami egyszerű, könnyen ellenőrizhető kritérium szerint értendő. Úgy is mondhatjuk, hogy mindig valamiféle „helyi", messze nem a teljes képet figyelembe vevő optimumot választunk, és reméljük, hogy ezek a választások összességükben jó megoldáshoz vezetnek. Ezt az egyszerűségre törekvést két nem teljesen független tényező motiválja. Egyrészt a teljes kép általában túl bonyolult ahhoz, hogy egy pillantásra megértsük. Másfelől egyszerűbb szempontok alapján könnyebben, gyorsabban tudunk választani. Láttunk szép példákat olyan esetekre, amikor a mohó megközelítés megalapozott – ide sorolhatók Dijkstra, Kruskal és Prim módszerei. Mindhárom módszernél megfigyelhető, hogy egyszerű mohó választások sorozatával érik el a kívánt megoldást.

Természetesen ennek, és a többi tervezési szemléletnek is megvannak a korlátai. A mohó módszer kritikájaként gyakran említik a következő metaforát: képzeljük el, hogy a célunk a Mount Everest megmászása. Azt a mohó megközelítést alkalmazzuk, hogy a következő lépésünket mindig abba az irányba tesszük, amerre legmeredekebb az emelkedés. Világos, hogy ily módon legtöbbször valami kis dombtetőn akadunk el. Mindez arra mutat, hogy a módszer alkalmazhatóságát, az általa elérhető eredmény minőségét körültekintően meg kell vizsgálni.

A gyorsrendezés meggyőzően példázza a **véletlent használó módszerek** erejét. Az algoritmus gyakorlati sikerének a kulcsa a partícionáló elem véletlen választása. Az ilyen módszerekkel – kiemelkedő fontosságuk miatt – részletesebben foglalkozunk ebben a fejezetben.

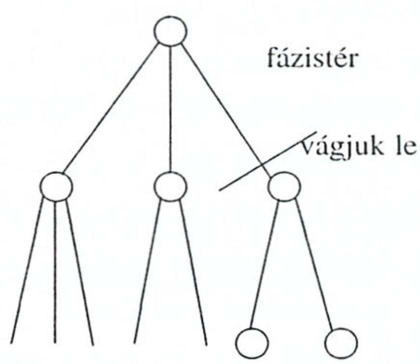
Eddig példaként csupa hatékonyan kezelhető problémát említetünk. A bemutatásra kerülő további technikák nehéz problémák kezelésére is alkalmasak. Mit várhatunk az ilyen esetekben? Szélsebes megoldásokat nem, hiszen nehéz feladatokról van szó. Például a tudomány mai állása szerint kevés a remény, hogy NP-nehéz feladatok megoldására igazán hatékony (polinom idejű) módszereket találjunk. Mégis, számos gyakorlati probléma megköveteli, hogy ilyen (vagy még nehezebb) feladatokra is adjunk valamilyen, természetesen minél jobb megoldó módszert. Lehetséges és gyakran értelmes célkitűzés az, hogy a kézenfekvő, minden lehetőséget végignéző (igen gyakran butának, bambának, stb. aposztrofált)

módszereknél jobbat találjunk. Másféle viszonyulást jelentenek a **közelítő módszerek**, melyek akkor jönnek szóba, ha a pontos megoldásnál kevesebbel is beérjük. Az **elágazás és korlátozás** elnevezésű módszert kifejezetten a nehéz feladatok esetén használják. A **dinamikus programozás** könnyű és nehéz problémákra egyaránt hasznos lehet. Végezetül itt tárgyaljuk a **prekondícionálás** módszerét. Ez leggyakrabban afféle munkaszervezési ötletnek tekinthető, amivel egy feladat egyszerre több példányának a megoldását lehet gyorsítani.

9.1. Elágazás és korlátozás

A módszer angol neve *branch-and-bound*. Olyan esetekben szokás alkalmazni, amikor a megoldandó feladat természetes módon vezet egy óriási gyökeres irányított fa teljes, vagy részleges bejáráshoz. A fát gyakran nevezik a feladat *fázisterének*. A fa általában olyan hatalmas, hogy nem érdemes (olykor nem is lehetne) egyszerre tárolni minden csúcsát. Vannak viszont szabályaink, amelyekkel egy csúcs leszármazottait elő tudjuk állítani. Ezt szokásos kifejezéssel generálásnak nevezzük. A feladat általában azt követeli, hogy a fa igen sok csúcsáról rendelkezzünk információval. Az elágazás és korlátozás módszere ezt az információgyűjtést igyekszik gazdaságosan megszervezni.

Első példaként nézzük egy sakkállás értékelésének a problémáját. A cél annak a meghatározása, hogy melyik fél – világos vagy sötét – áll jobban, és mi(k) a helyes lépés(ek). Képzeljünk el ehhez egy fát; ennek csúcsai sakkállások, megjelölve azzal, hogy melyik félen van a lépés sora. Egy csúcsból (állásból) él vezet a belőle egyetlen lépéssel megkapható állásokba. Több sakkozó program használja ezt a szemléletet. Az állás értékének megállapításához az alatta levő részfa csúcsait kellene értékelni. Ez azonban túl nagy, túlságosan időigényes vállalkozás lenne.



Az *elágazás* azt jelenti, hogy vesszük (generáljuk) az éppen vizsgált csúcs fiait, bizonyos esetekben egy korlátos mélységig a további leszármazottait is. A generált csúcsok vizsgálata alapján döntjük el, hogy merre menjünk tovább lefelé a

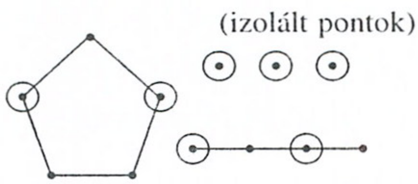
fában. Ebben fontos szerepet játszanak a *korlátozó heurisztikák*. Ezek a feladat tulajdonságaiból eredő megfontolások, amelyekkel igyekszünk minél több csúcsot és egyszerre azok leszármazottait is kizárni a további keresésből.

A sakk esetén az elágazás azt jelenti, hogy generáljuk a néhány lépésben elérhető állásokat, és azokat valahogy értékeljük. Korlátozó heurisztika lehet, hogy nem nézzük azokat az ágakat, ahol vezért vesztünk. Természetesen a komoly sakkprogramok többféle, a fentinél sokkal finomabb és bonyolultabb korlátozó heurisztikát használnak. A korlátozó heurisztikák alkalmazásával ágakat tudunk levágni a fáról, és csak az esélyes, a túlélő irányokkal foglalkozunk tovább.

A következőkben egy kevésbé bonyolult példát veszünk szemügyre: maximális független halmaz keresését gráfokban. A probléma eldöntendő változatáról tudjuk, hogy NP-teljes.

Legyen $G = (V, E)$ egy n pontú egyszerű irányítatlan gráf. A célunk az, hogy találjunk egy maximális méretű $H \subseteq V$ független halmazt. A kézenfekvő módszer az lenne, hogy végignézzük V összes részhalmazát, mindegyikről eldöntve, hogy független-e. Ez 2^n részhalmaz végigbogarászását jelentené. Próbáljuk meg ennél ügyesebben csinálni. Az esetek végignézésében alkalmazzuk a következő, igen egyszerű korlátozó heurisztikát:

Észrevétel: *Ha G-ben minden pont foka legfeljebb kettő, akkor a feladat lineáris időben megoldható, hiszen ekkor G izolált pontok, utak és körök diszjunkt uniója. Minden ilyen komponensben gyorsan találhatunk maximális független halmazt: komponensenként minden „második" pontot bevesszük a halmazba.*



Ezt használja az MF() algoritmus, aminek az inputja a G gráf.

MF(G)
1. Ha G-ben minden pont foka ≤ 2, akkor MF(G) az előbbi eljárás által adott maximális független halmaz, és a munkát befejeztük.
2. Legyen $x \in G$, $fok(x) \geq 3$.
 $S_1 :=$ MF($G \setminus \{x\}$)
 $S_2 := \{x\} \cup$ MF($G \setminus \{x$ és szomszédai$\}$).
3. Legyen S az S_1 és S_2 közül a nagyobb méretű, illetve akármelyik, ha $|S_1| = |S_2|$.
4. MF(G) $:= S$.

A módszer helyességének megértéséhez elég meggondolni, hogy egy maximális független halmaz vagy tartalmazza x-et, és ekkor a szomszédait biztosan nem (ilyen halmaz kerül S_2-be), vagy nem tartalmazza x-et (maximális méretű ilyen halmaz kerül S_1-be).

Nézzük ezután az eljárás költségét. Legyen $T(n)$ az MF(G)-n ($| V(G) | \leq n$) belüli MF hívások maximális száma, beleértve MF(G)-t magát is.

Állítás: *Van olyan c állandó, hogy* $T(n) \leq c\gamma^n$*, tetszőleges* n *természetes számra, ahol* γ *a* $\gamma^4 - \gamma^3 - 1 = 0$ *egyenlet pozitív gyöke* ($\gamma \approx 1,381$).

Bizonyítás: Legyen $t(n) := T(n) + 1$. Az eljárás szerkezetéből azonnal kapjuk, hogy $T(n) \leq T(n-1) + T(n-4) + 1$, ha $n > 4$. Innen adódik, hogy $t(n) \leq t(n-1) + t(n-4)$, ha $n > 4$. Ezután elég megmutatni, hogy $t(n) \leq c\gamma^n$ teljesül alkalmas c-vel, mert $T(n) < t(n)$. Indukciót használunk. A kívánt egyenlőtlenség $n < 5$-re alkalmas (elég nagy) c-vel elérhető. Ezután, ha $n \geq 5$, akkor alkalmazható az indukciós feltevés:

$$t(n) \leq t(n-1)+t(n-4) \leq c\gamma^{n-1}+c\gamma^{n-4} = c\gamma^{n-4}(\gamma^3+1) = c\gamma^{n-4}\gamma^4 = c\gamma^n.$$

Az utolsó előtti egyenlőségnél használtuk a γ definiáló egyenletét. □

A hívások előtti és utáni munka $O(n^d)$ alkalmas d-re. Ezt a költséget az MF(G) hívásra terheljük. Így becsülve az összes munka $O(n^d T(n)) = O(n^d\gamma^n) = O(1,381^n)$. Itt figyelembe vettük, hogy $\gamma < 1,381$. A kapott módszer tehát, bár a korlát itt is exponenciális, jóval gyorsabb, mint az összes eset végignézése. Ennél a példánál a fa csúcsainak az MF()-hívásokat tekinthetjük. Az MF(G) fiai a 2. lépésben megadott hívások. A G gráfhoz tartozó csúcsnak nincs további fia, ha az 1. lépésben végzünk (sikeres korlátozás), és két fia van különben.

Az elágazás és korlátozás módszerének másik alkalmazásaként vegyük szemügyre a 3 színnel való színezés feladatát. Az eldöntési probléma itt is NP-teljes. Mint előbb, legyen $G = (V, E)$ egy n pontú egyszerű irányítatlan gráf. Szeretnénk megadni G egy jó 3-színezését, ha egyáltalán létezik ilyen. A kézenfekvő módszer a csúcsok 3^n színezésének a végignézése lehetne. Korlátozó heurisztikát ad a következő észrevétel: ha kijelöltük az egyik színhez tartozó csúcsokat, mondjuk a pirosakat, akkor polinom időben ellenőrizhető, hogy ez a részleges színezés kiteljesíthető-e jó 3-színezéssé. Valóban, mindössze arról kell megbizonyosodni, hogy a piros pontok között nem fut él, továbbá, hogy a színezetlen pontok által feszített gráf kiszínezhető két színnel. Mindkét részfeladat legyűrhető polinom időben.

Feladat: Mutassuk meg, hogy az előbbi korlátozó heurisztika alkalmazásával a 3-színezés feladata megoldható $O(2^n n^c)$ időben, ahol $c > 0$ állandó.

9.2. Dinamikus programozás

A dinamikus programozás módszere gyakran használatos valamilyen numerikus paraméterektől függő érték optimumának a meghatározására. Lényege, hogy az optimumot (optimális megoldást) alkalmas kisebb részfeladatok optimumából (optimális megoldásából) próbáljuk előállítani. A dinamikus programozást használó algoritmusok vezérlési szerkezete gyakran emlékeztet egy *táblázat* szisztematikus (pl. sorról sorra haladó) kitöltésére. A táblázat kitöltését általában egy *rekurzív összefüggés* teszi lehetővé, ami alapján a bejárási sorrend szerint korábbi elemekből meghatározhatók a későbbiek. A kapott módszer költségét többnyire a kitöltendő táblázat mérete határozza meg. A rekurzív összefüggés megtalálásában sokszor a segítségünkre van az *optimalitás elve*, amivel a Bellman–Ford- és a Floyd-módszernél már találkoztunk.

A dinamikus programozásnak vannak erőteljes alkalmazásai a könnyű és nehéz problémák körében egyaránt. Működését néhány példán keresztül szeretnénk bemutatni.

1. Binomiális együtthatók számítása
Tegyük fel, hogy az $\binom{n}{k}$ binomiális együttható értékére vagyunk kíváncsiak. Lehetséges utat jelent a jól ismert

$$(*) \qquad \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

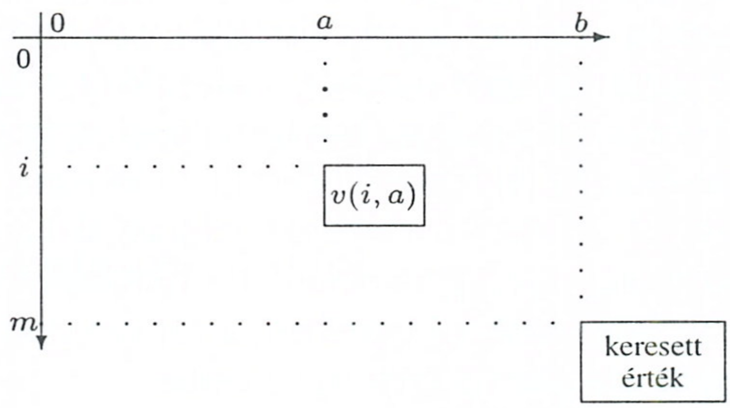
azonosság használata. Ennek segítségével a kisebb n értékektől a nagyobbak felé haladva adódnak a binomiális együtthatók: ha az összes $\binom{n-1}{j}$ értéket ismerjük ($0 \leq j \leq n-1$), akkor az $\binom{n}{k}$ alakú együtthatók egy-egy összeadással megkaphatók. Az elindulás lehetőségét az $\binom{m}{0} = \binom{m}{m} = 1$ értékek biztosítják. Tulajdonképpen a nevezetes Pascal-háromszög (egy háromszög alakú táblázat) kitöltéséről van szó. Az algoritmus csak összeadásokat használ, ezért gyakorlati szempontból is érdekes lehet olyan aritmetikai környezetben, ahol az összeadás sokkal gyorsabb, mint a szorzás.

A dinamikus programozás *alulról építkező*, a kisebb esetektől a nagyobbak felé menő stratégiája gyakran hatékonyabb alternatívát kínál a rekurzív eljárások felülről lefelé haladó felfogásával szemben. A binomiális együtthatók számítása jól mutatja ezt. A $(*)$ összefüggés alapján megírt rekurzív eljárás árnyoldala, hogy többször is kiszámítja ugyanazokat a (közbülső) binomiális együtthatókat. Például az $\binom{n}{k}$ számításakor az $\binom{n-2}{k-1}$ együtthatót kétszer is kiszámoljuk. A dinamikus programozás gondolatát követő módszer kiküszöböli ezeket az ismétlődéseket.

2. A Hátizsák probléma

Adottak az $s_1, \ldots, s_m$ súlyok, a b súlykorlát, a $v_1, \ldots, v_m$ értékek és a k értékkorlát. A kérdés, hogy van-e olyan $I \subseteq \{1, \ldots, m\}$ részhalmaz, melyre teljesül, hogy $\sum_{i \in I} s_i \leq b$ és $\sum_{i \in I} v_i \geq k$. Feltesszük még, hogy a szereplő mennyiségek mind pozitív egészek. A feladatról láttuk, hogy NP-teljes.

Itt most egy dinamikus programozást használó megoldást ismertetünk. Szeretnénk a feladatot visszavezetni kisebb hasonló problémákra. Ilyenkor gyakran segít, ha a feladatban megadott bizonyos paramétereket változónak tekintjük, és egy értelmes tartományban „futni hagyjuk". (Ez az ötlet erős eszköz a programozáselméletben, amikor ciklusinvariánst keresünk.) A mi esetünkben az m és a b lesznek ezek a paraméterek. Pontosabban fogalmazva legyen $v(i, a)$ a maximális elérhető érték az $s_1, \ldots, s_i$ súlyokkal, $v_1, \ldots, v_i$ értékekkel és a súlykorláttal megadott feladatra (mivel a maximális értéket keressük, nincs szükség értékkorlátokra). Ekkor $v(0, a) = v(i, 0) = 0$ tetszőleges a és i természetes számokra, és célunk a $v(m, b)$ mennyiség meghatározása, illetve annak eldöntése, hogy fennáll-e a $v(m, b) \geq k$ egyenlőtlenség. A feladat úgy is felfogható, hogy meg akarjuk határozni az $m + 1$ sorból és $b + 1$ oszlopból álló $[v(i, a)]$ táblázat (m, b) pozíciójú elemét.



A táblázat 0 indexű sorában, illetve oszlopában az értékek ismertek. Az érdekesebb helyeken levő számok meghatározásában segít a következő egyszerű összefüggés:

$$v(i, a) = \max\{v(i-1, a); v_i + v(i-1, a-s_i)\}.$$

Indoklásul megjegyezzük, hogy a jobb oldalon az első mennyiség az i-edik súlyt nem tartalmazó választások optimális értéke, a második pedig az i-edik súlyt tartalmazó választások optimális értéke (mindkét esetben a súlykorlát mellett). Itt is érvényesül az optimalitás elve: ha a $v(i, a)$ értéket adó kitöltésben az s_i súly szerepel, akkor a zsákban levő többi (az $s_1, s_2, \ldots, s_{i-1}$ közül kikerülő) súlynak optimális kitöltést kell adnia az $a - s_i$ súlykorláttal.

Az összefüggés alapján a táblázat kitölthető úgy, hogy vesszük rendre az $1, 2, \ldots, m$ indexű sorokat, ezeken belül pedig az a indexet növelve haladunk. A táblázatnak mb eleme van. A módszer logaritmikus költsége $O(bL)$, ahol L az input hossza. A módszer *nem polinomiális idejű*, mert b mérete $\lceil \log_2(b+1) \rceil$, az input hossza pedig

$$L = \sum_{i=1}^{m} (\lceil \log_2(s_i + 1) \rceil + \lceil \log_2(v_i + 1) \rceil) + \lceil \log_2(k+1) \rceil + \lceil \log_2(b+1) \rceil.$$

Másfelől már a táblázat mérete is legalább mb, ami L-hez képest exponenciálisan nagy is lehet. Ha viszont b nem túl nagy a többi input paraméterhez képest, akkor a módszer akár polinom idejű is lehet. Ezt most pontosabban is megfogalmazzuk.

Definíció: *A b egész* unáris ábrázolása*:* $0^b := 0 \cdots 0$ *(összesen b darab 0 egymás után).*

Definíció: *Egy feladat egy egész input paramétere* apró*, ha unárisan számítjuk bele az input hosszába.*

Ha a b paraméter apró, akkor az input méretéhez való hozzájárulása $|b|$ és nem $\lceil \log_2(b+1) \rceil$, mint a bináris megadás esetén. A b-t akkor érdemes aprónak tekinteni, amikor az unáris ábrázolása az inputban nem növelné meg az input méretének *nagyságrendjét*. Ez teljesül, ha $|b|$ felülről becsülhető az input más összetevői méretének egy polinomjával. Például a Hátizsák feladatnál ha $b \leq m^5$, akkor b-t szemlélhetjük úgy, mint egy apró paramétert. Ez természetesen nem jelenti azt, hogy b-t unárisan kell ábrázolnunk az inputban; arról van csupán szó, hogy a költségszámításnál az unáris hosszát vesszük figyelembe.

Következmény: *A Hátizsák probléma apró súlykorlát esetén megoldható polinom időben.*

Igazolásul elég annyi, hogy ekkor $L \geq b$, tehát a futási idő az input hosszának polinomjával becsülhető: $O(L^2)$.

Feladat: A Hátizsák probléma apró értékkorlát esetén megoldható polinom időben.

3. Legrövidebb utak gráfokban

Itt egy korábban megismert algoritmusra pillantunk ismét, felismerve rajta a dinamikus programozás gondolati jegyeit. Ez Floyd módszere az összes pontpár közötti távolság meghatározására: adott a C adjacencia mátrixával egy súlyozott élű

G irányított gráf, aminek csúcshalmaza $V = \{1, 2, ..., n\}$. Feltesszük, hogy G-ben nincs negatív összsúlyú irányított kör. A módszer $k = 1, 2, \ldots, n$-re sorban meghatározza az $F_k[i, j]$ értékeket, ahol az $F_k[i, j]$ a legrövidebb olyan $i \rightsquigarrow j$ utak hossza, melyeknek a közbülső pontjai az $1, 2, \ldots, k$ csúcsok közül valók. A számítás alapja az

$$F_k[i, j] := \min\{F_{k-1}[i, j], F_{k-1}[i, k] + F_{k-1}[k, j]\}$$

rekurzió. A kezdőértékeket a C mátrix adja: $F_0[i, j] := C[i, j]$. Itt is arról van szó, hogy az egyre összetettebb (egyre több közbülső pontot megengedő) optimumokat az egyszerűbbekből származtatjuk. Az algoritmus felfogható egy térbeli táblázat lapról lapra haladó kitöltésének. Legyen ugyanis $B[k, i, j] = F_k[i, j]$. Tulajdonképpen a B tömb értékeit határozzuk meg a $B[0, i, j]$ ($1 \leq i, j \leq n$) laptól indulva. Floyd módszerének elegáns, helytakarékos vonása, hogy ez kivitelezhető egyetlen n-szer n-es tömb alkalmazásával (amit az algoritmus leírásakor $F[i, j]$-vel jelöltünk).

Feladat: Adott az $A[1 : n, 1 : n]$ kétdimenziós Boole-tömb. Adjunk $O(n^2)$ uniform költségű módszert az A-beli legnagyobb csupa egyesből álló négyzetek egyikének a megkeresésére. Pontosabban: határozzuk meg a legnagyobb olyan $0 \leq k < n$ egészet, melyhez vannak olyan i, j indexek, hogy az $A[i : i + k, j : j + k]$ résztömb minden eleme egyes.

9.3. Közelítő algoritmusok

Ne ránts tőrt, ha egy pofon is megteszi.
SKÓT KÖZMONDÁS

Előfordul, hogy egy nehéz algoritmikus problémával kerülünk szembe, de elegendő a céljainkhoz a megoldás (tipikusan valamilyen optimum) elég jó közelítése. Bizonyos esetekben igen egyszerű és hatékony módszerekkel kaphatunk az optimálishoz közeli eredményt. Érdemes lehet közelítő módszert használni akkor is, ha a problémára van ugyan gyors módszer, de egy a céloknak megfelelő közelítést még gyorsabban ki lehet számítani. Lássunk ezután néhány példát!

1. Sok független él kiválasztása egy gráfból
Legyen adott éllistával az n pontú és e élű $G = (V, E)$ irányítatlan gráf. Szeretnénk minél nagyobb méretű $E' \subseteq E$ független élhalmazt találni. (Az E' élhalmaz

független, ha semelyik két E'-beli élnek sincs közös végpontja.) Ez kezelhető feladat. Egy maximális méretű E' a korábban használt terminológiánk szerint maximális párosítás, ami polinom időben található. A páros gráfok esetét részletesen tárgyaltuk a gráfalgoritmusokról szóló részben. Maximális párosítás (tetszőleges, tehát nem feltétlenül páros gráfban is) építhető ugyan polinom időben, de az ismert algoritmusok elég időigényesek. Előfordulhat másfelől, hogy elegendő egy nagy független élhalmaz, ami esetleg nem maximális. Szerencsét próbálhatunk például a következő mohó ötlettel:

Kezdetben legyen $E' = \emptyset$. Vegyünk egy $e \in E$ élet, tegyük E'-be, majd töröljük E-ből azokat az éleket, amelyeknek van e-vel közös végpontjuk. A megmaradó élek közül tegyünk egy f élet E'-be, majd töröljük az f-fel közös végpontú éleket, stb. egészen addig, amíg E el nem fogy.

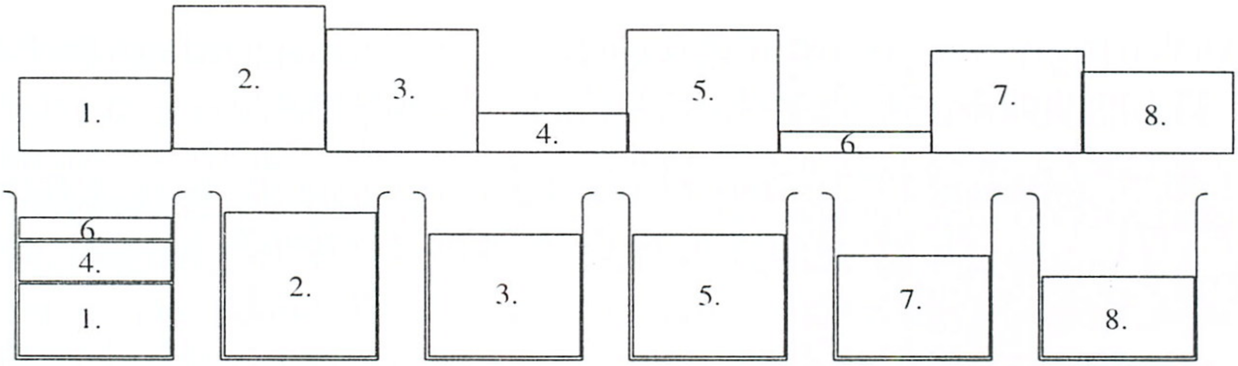
A fenti igen egyszerű módszer nagyon gyorsan, $O(n + e)$ költséggel megvalósítható. A kapott E' tovább már nem bővíthető független élhalmaz. Ha egy ilyen halmaz is megteszi, akkor felesleges az időigényesebb módszerekhez fordulni.

Feladat: Legyen a G-beli maximális független élhalmazok elemszáma k. Mutassuk meg, hogy a fenti algoritmus által kiválasztott E' élhalmaz mérete legalább $k/2$.

2. Ládapakolás

A következőkben a Ládapakolás feladat (angol nevén: Bin packing) néhány egyszerű és nevezetes közelítő megoldását mutatjuk be. Inputként adottak az $s_1, \ldots, s_m$ (racionális) súlyok, $0 \leq s_i \leq 1$. A cél a súlyok elhelyezése minél kevesebb 1 súlykapacitású ládába. Tudjuk, hogy a probléma döntési feladatként megfogalmazva NP-teljes.

Igen egyszerű és természetes közelítő algoritmus az FF-módszer (*first fit*, magyarul *első alkalmas*). Vegyünk először üres ládákat, és számozzuk meg őket az $1, 2, \ldots, k$ egészekkel. Ennyi előkészület után ismertethetjük a módszer általános lépését: tegyük fel, hogy az $s_1, \ldots, s_{i-1}$ súlyokat már elhelyeztük. Ekkor s_i kerüljön az első (legkisebb sorszámú) olyan ládába, amelybe még befér. Az ábra az FF-algoritmus működését illusztrálja.



Jelölje a Ládapakolás probléma egy I inputjára $OPT(I)$ az optimális (minimálisan elegendő), $FF(I)$ pedig az FF-módszer által eredményezett ládaszámot.

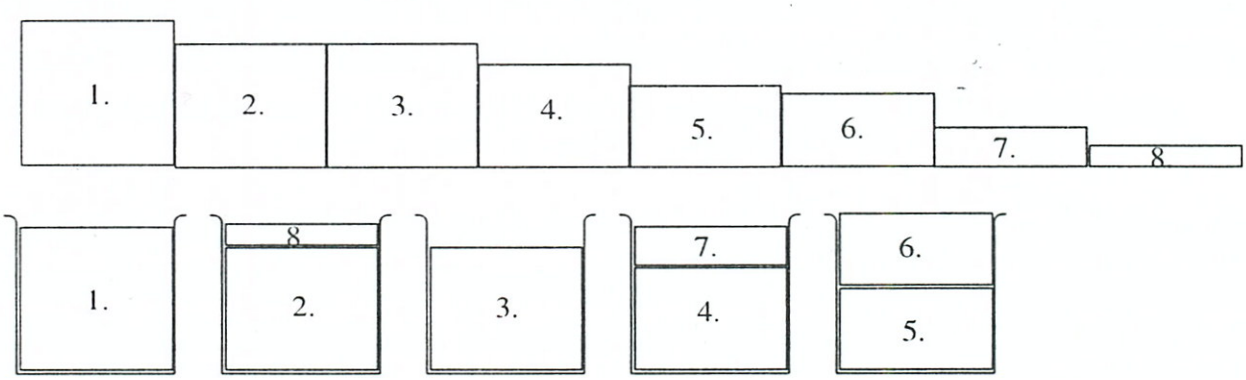
Világos, hogy $\lceil \sum_{i=1}^{m} s_i \rceil \leq OPT(I)$, továbbá $FF(I) \leq \lceil 2 \sum_{i=1}^{m} s_i \rceil$, hiszen az FF elhelyezésnél legfeljebb egy olyan használt láda lehet, amely nincs félig kitöltve. A két egyenlőtlenséget összefűzhetjük, figyelembe véve, hogy tetszőleges valós x-re $\lceil 2x \rceil \leq 2\lceil x \rceil$.

Következmény: *A probléma tetszőleges I inputjára teljesül, hogy* $FF(I) \leq 2OPT(I)$.

Valójában ennél sokkal több mondható (nem bizonyítjuk, nehéz).

Tétel (D. S. Johnson és munkatársai, 1976): *A probléma tetszőleges I inputjára teljesül, hogy $FF(I) \leq \lceil \frac{17}{10} OPT(I) \rceil$. Továbbá vannak tetszőlegesen nagy méretű I inputok, melyekre $FF(I) \geq \frac{17}{10}(OPT(I) - 1)$.*

Az FF tehát igen egyszerű és gyors módszer, amely eléri az optimum 17/10-ét. Egy kis javítással még jobb eredményt kaphatunk. Az FFD- (first fit decreasing, magyarul első alkalmas csökkenő) módszer receptje a következő: először rendezzük a súlyokat nem növő sorrendbe, utána alkalmazzuk az FF-módszert. Másképpen fogalmazva: az FF alkalmazásánál feltesszük, hogy $s_1 \geq s_2 \geq \ldots \geq s_m$. Az előző példa súlyai így eggyel kevesebb ládába bepakolhatók.



Általában érvényes a következő (nem bizonyítjuk, nagyon nehéz). Itt $FFD(I)$ jelöli az I inputon az FFD-módszer által adott ládaszámot.

Tétel (D. S. Johnson, 1973): *Tetszőleges I inputra teljesül, hogy $FFD(I) \leq \frac{11}{9}OPT(I) + 4$, és tetszőlegesen nagy méretű I inputok vannak, melyekre $FFD(I) \geq \frac{11}{9}OPT(I)$.*

Az FFD-módszerrel elérhetjük tehát az optimális érték mintegy $11/9$-ét. Az algoritmus rendkívül egyszerű, ugyanakkor jó közelítést ad. Megemlítjük még, hogy az FFD-nél valamivel jobb eredményt adó módszerek is vannak, ezek azonban egyszersmind jóval bonyolultabbak is.

3. Euklideszi utazó ügynök

Ez probléma az *Utazó ügynök* feladat egy változata. Az n pontú K_n teljes gráf élein adott a nemnegatív értékű d súlyfüggvény. Erre teljesül a háromszög-egyenlőtlenség: tetszőleges különböző u, v, w csúcsokra érvényes a

$$d(u, w) \leq d(u, v) + d(v, w)$$

egyenlőtlenség (az euklideszi feltétel). A cél egy minimális összsúlyú Hamilton-kör keresése.

Feladat: Mutassuk meg, hogy az *Euklideszi utazó ügynök* eldöntési változata NP-teljes. (A H nyelvet, a Hamilton-kört tartalmazó gráfok nyelvét redukáljuk erre a nyelvre. Legyen a G gráf a H egy inputja. A G éleinek a súlya legyen 1, a G komplementerébe tartozó éleké pedig legyen 2.)

Jó közelítő megoldást kaphatunk a következők szerint. Vegyük először a súlyozott K_n gráf egy T minimális összsúlyú feszítőfáját. Jelölje s a T fa költségét (súlyát). Egy ilyen T fa hatékonyan kapható Kruskal vagy Prim módszerével. Helyettesítsük a T éleit egy-egy pár ellentétesen irányított éllel: az (u, v) irányítatlan él helyett az $u \to v$ és a $v \to u$ éleket vesszük. Az így kapott irányított gráf legyen T'. A T'-ben minden pontnak a be-foka megegyezik a ki-fokával, tehát van zárt Euler-bejárása. Egy ilyen bejárás gyorsan számítható. Írjuk le ezután a K_n csúcsait egy olyan sorrendben, ahogy az Euler-séta során végiglátogatjuk őket:

$$(*) \qquad u = v_1, v_2, \ldots, v_{2n-2}, v_{2n-1} = u.$$

Ebből a sétából a kitérők levágásával fogunk Hamilton-kört kapni. Először is megjegyezzük, hogy a listán K_n minden csúcsa szerepel, és a listán szomszédos két csúcs között él fut T-ben. Az is igaz, hogy a T egy éle pontosan kétszer fordul elő mint a körsétán szomszédos csúcsok közötti él. Ebből következik, hogy a $(*)$

körséta összsúlya éppen kétszerese a T összsúlyának, azaz $2s$. Ezután a $(*)$ mentén haladva megadjuk a K_n egy F Hamilton-körét. Legyen a kiindulópont u. Elég megadni azt a sorrendet, ahogy a K_n pontjait F-ben az u pontból indulva végigjárjuk. Tegyük fel, hogy az $u = u_1, u_2, \ldots, u_k$ pontokat már kiválasztottuk, és $u_k = v_i$. Legyen $j > i$ a legkisebb olyan index, melyre teljesül, hogy v_j nincs az $u_1, u_2, \ldots, u_k$ pontok között. Ha van ilyen j, akkor legyen $u_{k+1} := v_j$. Ha nincs ilyen j, akkor szükségképpen $k = n$ és a munkát befejezzük az $u_{n+1} := u$ választással. Minden csúcs szerepelni fog F-ben, hiszen a $(*)$ lista tartalmazza a K_n összes pontját. Az u-t kivéve minden pont egyszer szerepel, tehát F tényleg Hamilton-kör.

Mit mondhatunk az F Hamilton-kör összsúlyáról? Ez a mennyiség legfeljebb akkora mint a $(*)$ séta összsúlya, hiszen – az előző jelöléseket megtartva – a háromszög-egyenlőtlenség miatt

$$d(u_k, u_{k+1}) \leq d(v_i, v_{i+1}) + d(v_{i+1}, v_{i+2}) + \cdots + d(v_{j-1}, v_j).$$

Az F összsúlya ezért legfeljebb $2s$. Mennyire lesz ez jó közelítése a minimumnak? Egy Hamilton-körből egy élet elhagyva feszítőfát kapunk, aminek az összsúlya legalább s. Ebből következően bármely Hamilton-kör súlya is legalább s. A módszerünk tehát kettes szorzó erejéig közelíti az optimális értéket.

Megjegyzés: A módszer lelke a $(*)$ bejárás konstrukciója, amit a T' gráf segítségével vittünk végbe. Ez a lépés szemléltethető úgy, hogy a T fát lerajzoljuk a síkba, majd az u csúcstól elindulva végigsétálunk az ágai mellett. Képzeljük el, hogy a bal kezünkben egy ecsetet tartunk, és ezt a séta során végighúzzuk a T élein, a leveleknél visszafordulva a „másik oldalra". A séta akkor ér véget, amikor minden él mindkét oldalát befestettük. Ekkor éppen az u pontba érünk vissza. A sorrend, ahogy a gráf csúcsait elérjük, megfelel a T' egy Euler-körútjának.

A vázolt fabejáró módszer neve *Lindström-bejárás*. Bizonyos csúcsokat esetleg többször is meglátogat, de igen hatékonyan megvalósítható. Elsősorban a következő tulajdonsága miatt használják belső memóriabeli listák kezelésére.

Feladat: Legyen T egy bináris fa, u a gyökere. Mutassuk meg, hogy az u-ból induló Lindström-bejárás megvalósítható lineáris idejű, konstans munkaterületet használó algoritmussal (így pl. akkor is, ha nem használhatunk vermet, és nem írhatunk a fa csúcsaiba).

9.4. Véletlent használó módszerek

> *És mikor előállatta volna a Sámuel az Izráelnek minden nemzetségit, találák ki sors által az Benjámin nemzetségét. Azután az Benjámin nemzetségét előállatá, mindent az ő háza népével, és találák kivenni az Mátri nemzetségének sorsát, és azok közül találák az Sault, az Kisnek fiát...*
> SÁMUEL ELSŐ KÖNYVE, 10: 20-21.

Itt olyan számításokról lesz szó, melyekben megengedünk valamiféle véletlen választásokat. Az ilyen algoritmusok furcsa sajátossága, hogy a bemenet és az algoritmus maga nem határozzák meg egyértelműen a tényleges lépéseket, a számítási időt, olykor magát a végeredményt sem. Mindezekért a bizonytalanságokért bőségesen kárpótol bennünket a véletlent használó (szokásos szakkifejezéssel: randomizált) módszerek hatékonysága és eleganciája.

Egy példával már találkoztunk. A gyorsrendezés ideje nagymértékben függ attól, hogy a partícionáló elemek mennyire vágják egyforma darabokra a tömböt. A partícionáló elemek véletlen választásával az esetek jó részében elég egyenletes vágást kapunk. Gyakorlati szempontból a gyorsrendezés randomizált változata a legjobb az ismert általános rendező módszerek közül. A randomizálás nem csak a könnyű feladatok megoldásakor jön szóba. Látni fogjuk, hogy így olyan problémák kezelésére kaphatunk hatékony módszereket, melyekre nem ismert gyors hagyományos algoritmus (Turing-gép, RAM program).

A randomizált módszerekkel szemben két érvet szokás felhozni. Az egyiket már érintettük: a bizonytalanság, a hiba lehetősége mintegy bele van tervezve ezekbe a módszerekbe. Mint látni fogjuk, ettől nem kell igazán félnünk. Gondoljunk arra, hogy a számítógépek, amelyeken a determinisztikus módszereink futnak, szintén nem mentesek a hiba lehetőségétől. Azt mondhatjuk, hogy egy bizonyos kis pozitív valószínűséggel hibásan működnek. A kérdés ezután már csak az, hogy le tudjuk-e szorítani erre a szintre a randomizált módszereink hibázási valószínűségét. (Sokkal ez alá menni nem érdemes, hiszen ezek a módszerek is a mi gyarló számítógépeinken futnak). A válasz a kérdésre sok érdekes esetben igenlő: a hibavalószínűség hatékonyan tetszőlegesen kicsivé tehető.

A másik ellenérv elvi szempontból súlyosabb. A randomizált módszerek elemzésénél, viselkedésük vizsgálatakor hallgatólagosan feltesszük, hogy valamiféle „igazán véletlen" bitek kellően hosszú sorozatával rendelkezünk, mégpedig általában nulla költséggel. Valójában nem világos, hogy mit tekinthetünk véletlen sorozatnak, és az még kevésbé, hogy nyerhetünk-e ilyet egyáltalán valamilyen természeti folyamatból. Nem célunk itt ezeket a filozófiába nyúló kérdéseket taglalni;

izgalmasak ugyan, de tudomásunk szerint kevés algoritmikus tanulsággal szolgálnak. Gyakorlati szempontból mindez sokkal biztatóbban néz ki: a randomizált módszerek nagyon jól működnek a rendelkezésre álló egyszerű, olcsó álvéletlen sorozatokkal. Ennek a széles körben megfigyelt kedvező jelenségnek az okát még nem érti a számítástudomány.

Az első példánk egy olyan feladat lesz, melyre nem ismert hatékony hagyományos algoritmus. Ez a probléma a polinomazonosságok tesztelése.

Probléma: Adott behelyettesítéssel egy n-változós $f \in \mathbb{Z}[x_1, \ldots, x_n]$ egész együtthatós polinom. Tudjuk, hogy $\deg f \leq d$. El akarjuk dönteni, hogy f azonosan nulla-e.

A feltétel, hogy f behelyettesítéssel adott, azt jelenti, hogy f-fel egyetlen dolgot tehetünk: értékeket adhatunk a változóinak, és ekkor valamilyen rendelkezésünkre álló eljárás megadja f értékét az adott behelyettesítésnél. Az eljárást úgy tekinthetjük, mint egy fekete dobozt. Bemenetként adunk neki egy egész komponensekből álló $\alpha = (\alpha_1, \ldots, \alpha_n)$ vektort; válaszul a doboz kiadja az $f(\alpha_1, \ldots, \alpha_n)$ értéket. Ez az első látásra furcsának tűnő helyzet könnyen előfordulhat. Megeshet például, hogy f olyan kifejezéssel adott, amit nem lehet gyorsan kifejteni (vagy más, céljainknak megfelelő módon átalakítani), ugyanakkor könnyű a változók adott értékeire f-et kiszámítani. Például tekintsük az

$$f(x_1, x_2, \ldots, x_{2n}) = (x_1 + x_2)(x_3 + x_4) \cdots (x_{2n-1} + x_{2n})$$

polinomot. Ezt kifejtve 2^n tagot kapunk, viszont egy helyettesítési érték n összeadással és $n-1$ szorzással megkapható. Hasonló a helyzet a következő, változókból álló determinánssal:

$$D = \det \begin{pmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & & \vdots \\ x_{n1} & \cdots & x_{nn} \end{pmatrix}.$$

A D-t kifejtve $n!$ tagot kapunk, ugyanakkor bármely helyettesítési értéke $O(n^3)$ aritmetikai művelettel megkapható (Gauss-elimináció). Mindkét polinom foka n, a behelyettesítés gyors és könnyű, a kifejtés nehéz, hiszen túl nagy a végeredményben a tagok száma.

Problémánkat megfogalmazhatjuk úgy is, hogy a d fok függvényében minél kevesebb behelyettesítéssel *tanút* szeretnénk arra, hogy $f \not\equiv 0$. Tanún egy $\alpha = (\alpha_1, \ldots, \alpha_n)$ egészekből álló vektort értünk, melyre $f(\alpha) \neq 0$.

A következő tétel annak pontos megfogalmazása, hogy ha $f \not\equiv 0$, akkor egy véletlenül választott α jó eséllyel tanú lesz. Ez szemléletesen nyilvánvaló. Képzeljük el az $n = 2$ esetet, amikor is f-nek két változója van. Ha $f \not\equiv 0$, akkor a

$\{(\beta_1, \beta_2) \in \mathbb{R}^2; f(\beta_1, \beta_2) = 0\}$ halmaz egy görbe a síkon. Világos, hogy a görbe a sík majdnem minden pontját elkerüli. Egy véletlen pont óriási eséllyel tanú lesz. A szokásos módon $Prob(A)$ jelöli az A esemény valószínűségét.

Tétel (J. Schwartz lemmája): *Ha* $\deg f \leq d$, *és* $\alpha_1, \ldots, \alpha_n$ *egyenletes eloszlású, egymástól független véletlen elemei az* $\{1, \ldots, N\}$ *számhalmaznak, akkor* $f \not\equiv 0$ *esetén* $Prob(f(\alpha) = 0) \leq \frac{d}{N}$.

Bizonyítás: Indukció n szerint. $n = 1$ esetén f-nek legfeljebb d gyöke van, így $Prob(f(\alpha_1) = 0) \leq \frac{d}{N}$. Legyen $n > 1$. Fejtsük ki f-et x_1 szerint, és legyen $y = (x_2, \ldots, x_n)$. Ezzel a jelöléssel

$$f(x) = h_0(y) + h_1(y)x_1 + \cdots + h_k(y)x_1^k,$$

ahol $k \leq d$ és $h_k \not\equiv 0$. Ha $f(\alpha) = 0$, akkor vagy az igaz, hogy $h_k(\alpha_2, \ldots, \alpha_n) = 0$, vagy pedig az, hogy $h_k(\alpha_2, \ldots, \alpha_n) \neq 0$ és $f(\alpha_1, \alpha_2, \ldots, \alpha_n) = 0$. Mit mondhatunk e két nemkívánatos esemény valószínűségéről?

Tekintetbe véve, hogy $\deg h_k \leq d - k$, az indukciós feltevés szerint $Prob(h_k(\alpha_2, \ldots, \alpha_n) = 0) \leq \frac{d-k}{N}$. Ha pedig $(\alpha_2, \ldots, \alpha_n)$ rögzített úgy, hogy $h_k(\alpha_2, \ldots, \alpha_n) \neq 0$, akkor $Prob(f(\alpha_1, \alpha_2, \ldots, \alpha_n) = 0) \leq \frac{k}{N}$, hiszen az $\alpha_2, \ldots, \alpha_n$, értékek behelyettesítése után kapott egyváltozós polinom foka k.

Annak a valószínűsége, hogy a két nemkívánatos esemény valamelyike bekövetkezik, legfeljebb a két esemény valószínűségének az összege, ami nem több, mint $\frac{d-k}{N} + \frac{k}{N} = \frac{d}{N}$. □

Következmény: *Az* $\{1, 2, \ldots, 2d\}$ *halmazból vett véletlen* n*-komponensű* α *vektor esetén* $Prob(f(\alpha) \neq 0) \geq 1/2$, *ha* $f \not\equiv 0$. *Ekkora halmazból választva tehát legalább* $1/2$ *valószínűséggel adódik tanú. Ha* t*-szer függetlenül választunk ilyen helyettesítést, akkor legalább* $1 - \frac{1}{2^t}$ *valószínűséggel kapunk tanút.* □

Ahhoz, hogy találjunk egy tanút, várhatóan 2 kísérlet elegendő. Ha az f polinom T időben kiértékelhető $\{1, 2, \ldots, 2d\}$-beli α_i értékekkel, akkor várhatóan $2T$ időben adódik tanú. Általában véve tT időköltséggel $1 - \frac{1}{2^t}$ valószínűséggel találhatunk tanút.

Térjünk itt vissza egy kicsit a randomizált módszerek megbízhatóságával kapcsolatos kifogáshoz! Képzeljünk el egy roppant megbízható számítógépet. Mondjuk olyat, ami ezer esztendőnyi folyamatos működés során csak egyszer hibázik. (Ennyi idő alatt csak történik valami, ami megzavarja: közeli villámcsapás, üstökös felbukkanása, tatárdúlás, stb.) Tegyük fel, hogy ez a hiba egy adott időintervallumba annak hosszával arányos eséllyel kerül. Ekkor annak a valószínűsége, hogy a hiba a következő ezredmásodpercben lép fel, nagyvonalúan becsülve is legalább

$(1/2)^{50}$. Az előző eljárást 60-szor ismételve a hibás döntés esélye ennél sokkal kisebbé tehető. A t alkalmas (és nem túl nagy) választásával tehát elérhetjük a hagyományos algoritmusok biztonságát.

Nézzük ezután a Schwartz-lemma egy alkalmazását. Legyen $G = (L, U; E)$ páros gráf, $L = \{l_1, \ldots, l_n\}$ és $U = \{u_1, \ldots, u_n\}$. Az $M = (m_{ij})$ n-szer n-es mátrixot az alábbi módon értelmezzük:

$$m_{ij} = \begin{cases} x_{ij} & \text{ha } (l_i, u_j) \in E, \\ 0 & \text{különben.} \end{cases}$$

Az M mátrix elemei tehát nullák és független változók. Annyi különböző változó szerepel, ahány éle van G-nek.

Tétel: *G-ben akkor és csak akkor van teljes párosítás, ha* $\det M \neq 0$.

Bizonyítás: A determináns egy tagja kifejtés után $\pm m_{1\pi(1)} m_{2\pi(2)} \cdots m_{n\pi(n)}$ alakú, ahol π az $1, \ldots, n$ számok egy permutációja. Ha egy ilyen tag nem 0, akkor $(l_i, u_{\pi(i)}) \in E$, $i = 1, \ldots, n$, így ezek az élek teljes párosítást adnak. Ha tehát G-ben nincs teljes párosítás, akkor $\det M = 0$. Ha viszont van G-ben teljes párosítás, akkor annak egy nem 0 kifejtési tag felel meg. Különböző kifejtési tagok nem ejthetik ki egymást, mert bármely kettőben van két különböző változó. Tehát ekkor $\det M \neq 0$. □

A $\det M$ polinomra alkalmazhatjuk az előbbieket. Véletlent használó (randomizált) módszert kapunk annak eldöntésére, hogy van-e G-ben teljes párosítás. Nézzük az eljárás költségét. Itt $\deg \det M = n$, az α_i elemeket az $\{1, 2, \ldots, 2n\}$ halmazból választhatjuk. A Schwartz-lemma szerint ha $\det M \not\equiv 0$, akkor egy helyettesítés legalább $(1/2)$ valószínűséggel tanút ad. Egy kiértékelés (Gauss-elimináció) $O(n^3)$ aritmetikai művelettel $(+, -, *, /)$ elvégezhető. A fellépő számok mérete a számítás alkalmas szervezése mellett legfeljebb $\log_2 n!(2n)^n$, ami $O(n \log n)$. A módszer tehát *várhatóan polinom időben*[1] eldönti, hogy van-e G-ben teljes párosítás.

Érdekességként megemlítjük az előző tétel általánosítását (bizonyítás nélkül) tetszőleges irányítatlan gráfokra.

[1] Az Olvasóban felmerülhet a kérdés, hogy miként viszonyul az itt bemutatott randomizált algoritmus hatékonysága a magyar módszeréhez. Bizonyítás nélkül megemlítjük, hogy – további ötletek alkalmazásával – a módszer várható futási ideje levihető $O(n^{2,38})$-ra. Ez sűrű gráfok esetén elvben versenyképes a König-módszer idejével, sőt a Hopcroft–Karp-algoritmuséval is. A gyakorlatban azonban erre a feladatra ma jobbnak tűnnek a gráfelméleti hátterű algoritmusok.

Tétel (Tutte tétele): *Legyen* $G = (V, E)$ *egy irányítatlan gráf,* $V = \{v_1, \ldots, v_n\}$. *Legyen* $T = (t_{ij})$ *a következő mátrix:*

$$t_{ij} = \begin{cases} x_{ij} & \textit{ha } (v_i, v_j) \in E \textit{ és } i < j, \\ -x_{ij} & \textit{ha } (v_i, v_j) \in E \textit{ és } i > j, \\ 0 & \textit{különben.} \end{cases}$$

A G *gráfban pontosan akkor van teljes párosítás, ha* $\det T \not\equiv 0$.

A páros gráfokra javasolt randomizált módszer minden nehézség nélkül működik az általános esetben is. Ekkor $\det T$ lesz a polinom, amihez tanút keresünk.

9.4.1. Az RP nyelvosztály

Köszönöm, kedves véletlen, fogadd hálás köszönetemet!
SØREN A. KIERKEGAARD: Vagy-vagy[2]

A hagyományos számítások hatékonyságának értelmezésében hasznosak voltak az idő- és tárkorlátokkal megadott nevezetes nyelvosztályok. Most abból szeretnénk rövid ízelítőt adni, hogy a randomizált számítások miként férnek ebbe a keretbe. Olyan nyelvosztályt definiálunk – ez lesz az RP osztály –, melynek a nyelvei hatékonyan felismerhetők randomizált (véletlent használó) algoritmusokkal.

Definíció: *Az* $L \subseteq I^*$ *nyelv az* RP *nyelvosztályba tartozik, ha van olyan* $L_1 \in$ P *nyelv és* $c \geq 0$ *állandó, hogy*
(i) $L = \{x \in I^*, \textit{és van olyan } y \in I^* \textit{ szó, melyre } |\, y \,|=|\, x \,|^c \textit{ és } (x, y) \in L_1 \}$,
(ii) ha $x \in L$, *akkor az* $|\, x \,|^c$ *hosszú* $y \in I^*$ *szavaknak legalább a felére teljesül, hogy* $(x, y) \in L_1$.

Vegyük észre, hogy az (i) tulajdonság éppen azt fejezi ki, hogy $L \in$ NP. Az RP-beli nyelvek tehát egyben NP-beliek is: RP $\subseteq$ NP. Ha viszont az L nyelv polinom időben felismerhető, akkor az $L_1 = \{(x, 1); x \in L\}$ nyelvvel és a $c = 0$ állandóval teljesülnek a definíció kikötései, vagyis ekkor $L \in$ RP is igaz. A P $\subseteq$ RP $\subseteq$ NP tartalmazási relációkról azt gondoljuk, hogy valódiak; széles körben elfogadott sejtés szerint az RP nem esik egybe sem a P, sem pedig az NP nyelvosztállyal.

Legyen $L \in$ RP. Az L_1 nyelv és egy neki megfelelő polinom idejű algoritmus ismeretében az L felismerésére a következő egyszerű randomizált (véletlent

[2]Johannes szavai amikor hosszú keresgélés után végre rábukkant Cordéliára.

használó) algoritmus kínálkozik. Legyen $x \in I^*$ egy input szó. El kell döntenünk, hogy $x \in L$ teljesül-e. Válasszunk egy véletlen $y \in I^*$ szót, melynek hossza $|x|^c$, és döntsük el az $((x, y) \in L_1?)$ kérdést. Ha a válasz igenlő, akkor $x \in L$, ellenkező esetben a következtetésünk: „valószínűleg $x \notin L$". Vegyük észre, hogy az (i) feltétel miatt az igenlő válasz mindig helyes. Ha van tanú, akkor az x szó az L nyelvbe tartozik. A nemleges válasz lehet hibás. Előfordulhat, hogy $x \in L$ teljesül ugyan, de rossz tanújelölt akadt a kezünkbe. Az (ii) feltétel szerint azonban ennek a balszerencsének a valószínűsége legfeljebb $1/2$. Nevezzük az itt vázolt módszert RP-tesztnek. A hibázás valószínűsége tetszőlegesen kicsivé tehető a már megismert módon: t darab függetlenül választott y kipróbálása után legfeljebb $\frac{1}{2^t}$ a téves következtetés valószínűsége. Mivel $L_1 \in$ P és y hossza az x hosszában polinomiális, egy ilyen teszt polinom időben elvégezhető. Az $(x \in L?)$ kérdés tehát várhatóan polinom időben megválaszolható.

Megjegyzés: A randomizált módszerek költségének a tanulmányozásakor nem vettük számításba a „véletlen választások" költségét. A gyakorlatban ezek a választások álvéletlen objektumok (számok, bitek) generálását jelentik. Itt csak megemlítjük, hogy vannak gyors módszerek, melyek elég jó (megfigyelt) viselkedésű álvéletlen objektumokat eredményeznek.

Igen érdekesek a $Las\ Vegas :=$ RP $\cap$ coRP osztályba tartozó nyelvek. Ha $L \in Las\ Vegas$, akkor definíció szerint $L \in$ RP és $L' = I^* \setminus L \in$ RP. Nézzük, mi történik, ha egy $x \in I^*$ bemenő szóra lefuttatunk egy-egy RP-tesztet mind az $(x \in L?)$, mind pedig az $(x \in L'?)$ kérdésre. Ezt megtehetjük, mert L és L' is RP-beli nyelv. Ezek közül csak az egyik adhat igenlő eredményt, hiszen x pontosan az egyik nyelvnek eleme. (Az (i) feltétel miatt az RP-teszt nem tud úgy lódítani, hogy a nyelvbe nem tartozó szóról azt állítja, hogy benne van.) Ha valamelyik tesztre *igen* a válasz, akkor készen vagyunk. Mi van akkor, ha egyik teszt sem adott igenlő választ? Az (ii) feltételt alkalmazva az L és L' közül arra, amelyiknek x eleme, látjuk, hogy ennek a valószínűsége legfeljebb $1/2$. Két *nem* esetén tehát arra következtethetünk, hogy a tesztek nem jártak eredménnyel. A tesztek ismétlésével ennek a valószínűsége gyorsan lecsökkenthető. A lényeges különbség a szimpla RP-teszthez képest, hogy itt sosem kapunk helytelen eredményt. Ha egyik teszt sem adott igenlő választ, akkor úgy vehetjük, hogy tovább kell próbálkoznunk az igazság kiderítésével. Az itt vázolt *Las Vegas-teszt* becsületes abban az értelemben, hogy sosem ad hamis választ a kérdésre. A Las Vegas-teszt jellemzőit így foglalhatjuk össze: *gyors (polinom idejű), nagy valószínűséggel válaszol az $(x \in L?)$ kérdésre, és a válasza mindig helyes.*

Az RP nyelvosztály definíciójában az (ii) feltételt úgy is fogalmazhatjuk, hogy ha $x \in L$, akkor az $|x|^c$ hosszú szavak közül választott véletlen y szó legalább

(1/2) valószínűséggel tanú lesz. Legyen p egy tetszőleges, rögzített, nem szélsőséges valószínűség, azaz legyen $0 < p < 1$. Az RP-beli nyelv definíciójában cseréljük ki az (ii) feltételt a következőre:

(ii') Ha $x \in L$, akkor egy véletlenül választott $| x |^c$ hosszú $y \in I^*$ szóra legalább p valószínűséggel igaz, hogy $(x, y) \in L_1$.

Feladat: Mutassuk meg, hogy ha a definícióban az (ii) feltétel helyett az (ii') feltételt használjuk, akkor is éppen az RP-beli nyelveket kapjuk.

Megemlítjük még, hogy a randomizált módszerek számítási ereje nagyobb a Turing-gépekénél. Tegyük fel ugyanis, hogy a (binárisan) adott n egész bemenetre találnunk kell egy olyan $x \in I^*$ szót, melyre $C(x) \geq n/2$ (itt $C(x)$ az x szó Kolmogorov bonyolultsága). Véletlen választással könnyen célt érünk: az olyan n hosszú x szavak száma, melyekre $C(x) \leq n - 2$, kisebb mint 2^{n-1}, tehát egy véletlenül választott n-hosszú x szóra legalább $1/2$ valószínűséggel teljesül, hogy $C(x) > n - 2$.

Másfelől nincs olyan M TG, mely az n bemenetre olyan $M(n)$ szót ad, melyre $C(M(n)) \geq n/2$. Ellenkező esetben az invariancia-tételből $C(M(n)) \leq C_M(M(n)) + c \leq \lceil \log_2(n + 1) \rceil + c$. Ezek összevetéséből pedig $n/2 \leq \lceil \log_2(n + 1) \rceil + c$ adódna, ami képtelenség, ha n elég nagy.

9.4.2. Prímtesztelés

Tegyük fel, hogy bemenő adatként adott (binárisan) egy m páratlan egész; szeretnénk eldönteni, hogy m prímszám-e. A feladat az elvi érdekességén túl gyakorlati szempontból is fontos. A biztonságos kommunikáció céljait szolgáló algoritmusok gyakran használnak igen hosszú – olykor többszáz-jegyű – prímeket. Hamar meggyőződhetünk róla, hogy ilyen nagy m-nél a kézenfekvő (exponenciális idejű) módszer, hogy m-et sorban elosztjuk a $\sqrt{m}$-nél kisebb természetes számokkal, nem fér bele az időnkbe.

A feladat tehát a Π nyelv felismerése. Erről eddig annyi derült ki, hogy jól karakterizált: $\Pi \in$ NP $\cap$ coNP. Azt is megemlítettük, hogy eddig nem találtak a megoldására polinom idejű módszert.

A prímszámok felismerésére szolgáló eljárásokat *prímteszteknek* szokás nevezni. Itt egy hatékony randomizált prímtesztet szeretnénk bemutatni. A módszer valójában egy RP-teszt lesz a prímtulajdonság komplementerének, az *összetettségnek* a felismerésére.

Először egy olyan módszert ismertetünk, ami szemléletes ugyan, de van egy kis fogyatékossága. Mint látni fogjuk, ez a fogyatékosság a módszer finomításával kiküszöbölhető. Az algoritmus (Fermat-teszt) a kis Fermat-tételen alapul. Az egyetlen bemenő paramétere a binárisan leírt $m > 2$ páratlan egész.

Fermat-teszt (m)
1. Válasszunk egy véletlen a egészet az $[1, m)$ intervallumból.
2. Ha $a^{m-1} \equiv 1 \pmod m$, akkor a válasz „m valószínűleg prím", különben a válasz „m összetett".

A teszt gyorsan ($\log_2 m$-ben polinomiális időben) végrehajtható: az $a^{m-1} \pmod m$ hatványt a korábban említett *gyors hatványozással* kaphatjuk meg hatékonyan.

Vegyük észre, hogy ha az eredmény az, hogy m összetett, akkor ez biztosan helyes. Ezt a tényt a tanúsítja, hiszen a 2. lépésben kiderül, hogy a-ra nem teljesül a Fermat-kongruencia. Ezért ekkor azt mondjuk, hogy a az m összetettségének *Fermat-tanúja*.

Példa: Legyen $m = 21 = 7 \cdot 3$ és $a = 2$. Ekkor a az m Fermat-tanúja, hiszen $2^{20} \equiv 4 \pmod{21}$.

A Fermat-tanúk számáról szól a következő állítás.

Állítás: *Ha m-nek van olyan a Fermat-tanúja ($1 \leq a < m$ és $a^{m-1} \not\equiv 1 \pmod m$), melyre $lnko(a, m) = 1$, akkor az $[1, m)$ intervallum egészeinek legalább a fele Fermat-tanú.*

Az állítást úgy is értelmezhetjük, hogy ha m összetettségének van olyan a Fermat-tanúja, melyre $lnko(a, m) = 1$, akkor a teszt legalább $1/2$ valószínűséggel talál Fermat-tanút.

Bizonyítás: Ha $lnko(a, m) > 1$, akkor a nyilván Fermat-tanú. Elég ezért látni, hogy a redukált maradékosztályok (amelyekre teljesül, hogy $lnko(a, m) = 1$) legalább fele Fermat-tanú. Legyen G a mod m redukált maradékosztályok halmaza és

$$H := \text{nem Fermat-tanúk} = \{b \in G;\ b^{m-1} \equiv 1 \pmod m\}.$$

Legyen $a \in G$ egy (a feltételünk szerint létező) Fermat-tanú, és tekintsük a $Ha := \{ba;\ b \in H\} \subseteq G$ halmazt. Ha b és b' két különböző maradékosztály, akkor $ba \not\equiv b'a \pmod m$, mert a és m relatív prímek. Ebből arra jutunk, hogy $|H| = |Ha|$.

Másfelől $Ha \subseteq G \setminus H$, hiszen egy H-beli maradékosztályt Fermat-tanúval szorozva ismét tanút kapunk. Mindezek alapján

$$|H| = |Ha| \leq |G \setminus H|,$$

azaz $|H| \leq \frac{1}{2} |G|$. Tehát a G elemeinek legalább a fele Fermat-tanú. □

A módszer hátulütője, hogy vannak olyan *m összetett* számok, melyekre egyáltalán nincs olyan a Fermat-tanú, melyre $lnko(a, m) = 1$. Ezek az úgynevezett

pszeudoprímek, vagy Carmichael-számok. Ilyen például az $561 = 3 \cdot 11 \cdot 17$. Újabb eredmény (W. R. Alford, A. Granville, C. Pomerance, 1992), hogy végtelen sok ilyen szám van. Hogyan ellenőrizhetjük, hogy 561 tényleg egy Carmichael-szám? Emlékeztetünk itt a kínai maradéktételre.

Tétel (kínai maradéktétel): *Legyenek* $m_1, \ldots, m_k$ *páronként relatív prím,* $a_1, \ldots, a_k$ *tetszőleges egészek. Ekkor az*

$$x \equiv a_1 \pmod{m_1}, \ldots, x \equiv a_k \pmod{m_k}$$

kongruencia rendszernek van egész x *megoldása. Az* x *megoldás modulo* $m_1 m_2 \cdots m_k$ *egyértelműen meghatározott.*

Visszatérve példánkhoz: meg kell mutatni, hogy ha $lnko(a, 561) = 1$, akkor $a^{560} \equiv 1 \pmod{561}$. A kínai maradéktétel szerint elég, ha az $a^{560} \equiv 1 \pmod{17}$, $a^{560} \equiv 1 \pmod{11}$ és $a^{560} \equiv 1 \pmod{3}$ kongruenciákat igazoljuk, feltéve, hogy a relatív prím a szóban forgó modulushoz. Nézzük meg az elsőt; a többi hasonlóan kezelhető. Mivel $560 = 16 \cdot 35$,

$$a^{560} \equiv a^{16 \cdot 35} \equiv \left(a^{16}\right)^{35} \equiv 1^{35} \equiv 1 \pmod{17}.$$

Az utolsó előtti kongruencia a kis Fermat-tétel következménye.

Érdemes megjegyezni azt a módot, ahogy itt a kínai maradéktételt használtuk. Modulo $m_1 m_2 \cdots m_k$ számítás helyett elegendő volt modulo m_i $(1 \le i \le k)$ számolni. Ezt az ötletet széles körben alkalmazzák a számítógépes aritmetikában, különösen a nagy pontosságú számítások területén.

E kis kitérő után kanyarodjunk vissza a prímteszteléshez. A Fermat-teszt fogyatékossága, hogy ha m Carmichael-szám, akkor m-et a teszt nagy eséllyel prímnek deklarálja. Most a Fermat-teszt olyan javítását ismertetjük, ami kiküszöböli ezt a problémát. A módszert *Rabin–Miller-tesztnek* nevezik.

Definíció: *Legyen* m *egy páratlan természetes szám. Írjuk fel* $m-1$*-et* $m - 1 = 2^k n$ *alakban, ahol* n *páratlan. Az* $1 \le a < m$ *egész* Rabin–Miller-tanú *(*m *összetettségére), ha az*

$$a^n - 1,\ a^n + 1,\ a^{2n} + 1, \ldots,\ a^{2^{k-1}n} + 1$$

számok egyike sem osztható m*-mel.*

Tétel: *Ha* m *prím, akkor* m*-hez nincs Rabin–Miller-tanú.*

Bizonyítás: Legyen a egy tetszőleges egész az $[1, m)$ intervallumból. Az

$$a^{m-1} - 1 = (a^n - 1)(a^n + 1)(a^{2n} + 1) \cdots (a^{2^{k-1}n} + 1)$$

azonosságot használjuk. Mivel m prím, a kis Fermat-tétel szerint m osztja a bal oldalon álló számot. Ugyancsak m prímvoltából következik, hogy m ekkor osztja a jobb oldal valamelyik tényezőjét, ami azt jelenti, hogy a nem Rabin–Miller-tanú. □

Az is látszik az azonosságból, hogy ha a nem Rabin–Miller-tanú, akkor nem lehet Fermat-tanú sem. Ugyanis ha a jobb oldali számok valamelyike osztható m-mel, akkor a bal oldal is osztható m-mel. Ezt pozitívba fordítva úgy is mondhatjuk, hogy egy Fermat-tanú egyben Rabin–Miller-tanú is. Az új definíció tehát tágítani igyekszik a lehetséges tanúk körét. A következő állítást úgy is olvashatjuk, hogy ez a bővítés sikerült: ha m összetett, akkor ennek a ténynek sok Rabin–Miller-tanúja van.

Tétel: *Ha m összetett, akkor az $1 \leq a < m$ feltételt teljesítő a egészeknek legalább a fele Rabin–Miller-tanú.*

A bizonyítást mellőzzük; csak annyit jegyzünk meg, hogy a lényeg annak az igazolása, hogy van egyáltalán m-hez relatív prím Rabin–Miller-tanú. Utána a Fermat-tesztnél látott érveléssel következik, hogy sok tanú van. A Rabin-Miller összetettségi teszt a Fermat-teszt kézenfekvő módosítása: Fermat-tanú helyett Rabin–Miller-tanút választunk.

$RM(m)$
1. Írjuk fel $m - 1$-et $m - 1 = 2^k n$ alakban, ahol n páratlan.
2. Válasszunk egy véletlen a egészet az $[1, m)$ intervallumból.
3. Ha az $a^n - 1$, $a^n + 1$, $a^{2n} + 1, \dots,$ $a^{2^{k-1}n} + 1$ számok egyike sem osztható m-mel, akkor megállunk azzal a válasszal, hogy „m összetett", különben megállunk azzal a válasszal, hogy „m valószínűleg prím".

Várhatóan 2 véletlen a érték választása után kapunk egy Rabin–Miller-tanút, ha m összetett. Ha t kísérlet után sem adódik Rabin–Miller-tanú, akkor m-et legfeljebb $\frac{1}{2^t}$ tévedési valószínűséggel prímnek nyilváníthatjuk. Az RM-teszt, ugyanúgy, mint a Fermat-teszt, gyors hatványozással valósítható meg hatékonyan.

Következmény: *Az összetett számok nyelve az* RP *osztályban van.*

Bizonyítás: Legyen

$$L_1 = \left\{ (m,a) \;\middle|\; \begin{array}{l} m > 2,\, 1 \le a < m \text{ egészek, és ha } m \text{ páratlan,} \\ \text{akkor } a \text{ egy } m\text{-hez tartozó Rabin–Miller-tanú} \end{array} \right\}$$

és $c = 1$. Ezekkel a választásokkal teljesülnek az RP-beliség (i) és (ii') feltételei, utóbbi a $p = (1/4)$ valószínűséggel. Legyen ugyanis n az m bitjeinek száma. A tételből következik, hogy a legfeljebb n bittel leírható egészek legalább $1/4$-e Rabin–Miller-tanúja lesz m összetettségének. □

Könnyű meggondolni, hogy a módszer éppen az L_1 nyelven alapuló RP-teszt az összetett számok felismerésére. Sokkal mélyebb eredmény (L. A. Adleman, M. D. Huang, 1987, a bizonyítás másfélszáz oldalnyi és pogányul nehéz), hogy $\Pi \in$ RP is teljesül. Maga a teszt polinom idejű ugyan, de olyan bonyolult, hogy a gyakorlatilag még érdekes méretű inputokon nem érdemes futtatni. Ezért prímtesztelésre inkább összetettségi teszteket használnak. Ezek között a Rabin–Miller-teszt elég jónak számít.

A két eredmény együtt azt jelenti, hogy $\Pi \in Las\ Vegas$. És még ennél is cifrább a helyzet. Egy sokat tanulmányozott, de mindeddig nem igazolt számelméleti hipotézisből (az ún. általánosított Riemann-sejtés) következik, hogy ha m összetett, akkor van olyan a Rabin–Miller-tanú is, melyre $a \le 2\log^2 m$ (E. Bach, 1990). Ha tehát a sejtés igaz, akkor $\Pi \in$ P, hiszen ekkor elég tanút keresni az első $2\log^2 m$ természetes szám között.

9.4.3. Nagy prímszám keresése

A későbbiekben szó lesz olyan módszerekről, amelyek nagy prímszámokat használnak. Itt megmutatjuk, hogy véletlen választással könnyen találhatunk ilyeneket.

Probléma: bemenő adatként adott egy n természetes szám. Keressünk egy n-jegyű prímszámot.

A módszer igen egyszerű: válasszunk egy véletlen p egészet a $[2^{n-1}, 2^n - 1]$ intervallumból. Az RM-teszttel ellenőrizzük, hogy p prím-e, mondjuk 2^{-60} hibavalószínűséggel. Ez legfeljebb 60 véletlen a kipróbálását jelenti. Ha kiderül, hogy p nem prím, akkor új p-t választunk.

Várhatóan hány p-t kell választani? Egy $x > 1$ valós számra jelölje $\pi(x)$ az $[1, x]$ intervallumba eső prímek számát. A prímszámtétel szerint, $\pi(x) \sim \frac{x}{\log x}$, ha $x \to \infty$. Innen könnyen adódik, hogy a számunkra érdekes intervallumban a prímek száma $\approx \frac{(\log_2 e)2^{n-1}}{n}$, ha n elég nagy. Annak a valószínűsége ezért, hogy egy véletlen p prím lesz, körülbelül $\frac{\log_2 e}{n}$, tehát $O(n)$ darab véletlen p kipróbálásával

várhatóan találunk egyet. A módszer várható költsége polinomiális az *eredmény hosszához mérve*.

9.5. Prekondícionálás

GUCUACGGCCAUACCACCUGAACGCGCCCGAUCUCGUCUG
AUCUCGGAAGCUAAGCAGGGUCGGGCCUGGUUAGUACUUG
GAUGGGAGACCGCCUGGGAAUACCGGGUGCUGUAGGCUU
Az emberi $5S$ rRNS genetikai kódja

Képzeljük el, hogy egy adott P algoritmikus problémának több példányát, mondjuk $i_1, i_2, \ldots, i_k$-t kell egy feladat részeként megoldanunk. A kézenfekvő megközelítés az lenne, hogy veszünk egy jó algoritmust a P megoldására, amit sorra lefuttatunk az $i_1, i_2, \ldots, i_k$ bemenetekkel. Sok esetben ennél számottevően hatékonyabb módszert kaphatunk, ha először egy kis előkészítő munkát végzünk – szokásos kifejezéssel *prekondícionáljuk* P-t. A prekondícionálás olyan előkészítő tevékenység, aminek az eredménye többször felhasználható. Bizonyos feladatoknál az előkészítő munka extra költsége több példány esetén megtérül, sőt az összköltség csökken. Gyakran egész egyszerűen arról van szó, hogy az $i_1, i_2, \ldots, i_k$ megoldásainak közös lépéseit kiemeljük, és előre elvégezzük. Ebben az értelemben a prekondícionálás hatékony munkaszervezési fogás. Előzetesen elvégezzük azokat a teendőket, amelyek a konkrét bemenetektől viszonylag függetlenek.

Az ilyen módszerek tervezésekor figyelnünk kell az előkészítő fázis költségeire is. Csak akkor érdemes belevágni, ha az $i_1, i_2, \ldots, i_k$ megoldásának plusz az előkészítésnek a költsége versenyképes a kézenfekvő megoldáshoz viszonyítva. Ennek mérlegelésekor érdemes meghatározni k-nak azt a nagyságrendjét, amitől kezdve már gazdaságos a módszer. Ennyi általánosság után nézzünk néhány példát:

1. Ősiség fákban

Tegyük fel, hogy T gyökeres, felfelé irányított fa (aminek az élei a gyökér felé mutatnak). A fa mint bemenet a csúcsaival és az apamutatókkal adott. A feladat T-beli (v, w) csúcspárokra eldönteni, hogy w leszármazottja-e v-nek. Egyetlen pár esetén a költségigény (uniform) lehet cn is (ahol n a T csúcsainak száma és $c > 0$ egy állandó). Ugyanakkor alkalmas $O(n)$ költségű prekondícionálás után minden egyes kérdés $O(1)$ költséggel kezelhető.

Egy lehetséges ötlet, hogy minden $v \in T$ csúcs mellé feljegyezzük a v T-beli $pre(v)$ preorder, illetve $post(v)$ postorder bejárás szerinti sorszámát. Nyilvánvaló

ugyanis, hogy a $v, w \in T$ csúcsokra $pre(v) \leq pre(w) \Leftrightarrow v$ őse w-nek vagy v a w-től balra van T-ben. Ugyanígy $post(v) \geq post(w) \Leftrightarrow v$ őse w-nek, vagy v a w-től jobbra van T-ben. A két tényből adódik, hogy v őse w-nek pontosan akkor, ha $pre(v) \leq pre(w)$ és $post(v) \geq post(w)$.

A kétfajta számozás $O(n)$ időben megkapható. A számok ismeretében egy (v, w) párra a kérdés két összehasonlítással megválaszolható. Ha a bemenetként adott párok száma (a bevezetőbeli k érték) nincs konstans korlát alatt, akkor érdemes ezt a bonyolultabb módszert használni. Például ha a párok száma $k = \log n$, akkor a kézenfekvő módszer költsége $cn \log n$ is lehet; a prekondícionált időigény viszont $O(n + \log n) = O(n)$.

2. Polinom ismételt kiértékelése

Adott az $f(x) = x^n + a_1 x^{n-1} + \cdots + a_n \in \mathbb{Z}[x]$ polinom, és a $b_1, b_2, \ldots, b_k$ egészek. Célunk az $f(b_1), \ldots, f(b_k)$ értékek kiszámítása minél kevesebb szorzással.

Mennyibe kerül itt a kézenfekvő módszer? Először kiszámítjuk a $b_i^2, b_i^3, \ldots, b_i^n$ hatványokat, ($n-1$ szorzás), majd az $a_{n-1}b_i, a_{n-2}b_i^2, \ldots, a_1 b_i^{n-1}$ számokat ($n-1$ szorzás). Összesen ez $2k(n-1)$ szorzás.

Javítást jelent a *Horner-elrendezés* alkalmazása: $f(x)$ felírható

$$f(x) = (\cdots(((x + a_1)x + a_2)x + a_3)\cdots)x + a_n$$

alakban (összesen $n-1$ zárójelpár van). Így végezve a számítást, $f(b_i)$-t $n-1$ szorzással kapjuk. Az összköltség $k(n-1)$ szorzás, ami fele az előzőnek. A Horner-elrendezés szerinti alakra való átalakítást is felfoghatjuk prekondícionálásnak.

Ezután egy még finomabb prekondícionálást használó megoldást mutatunk be. Az egyszerűség kedvéért tegyük fel, hogy $\deg f = n = 2^t - 1$, valamely t természetes számra. Írjuk fel f-et

$$f(x) = (x^{\frac{n+1}{2}} + a)f_1(x) + f_2(x)$$

alakban, ahol $a \in \mathbb{Z}$, az f_1, f_2 egész együtthatós, 1 főegyütthatós polinomok, és $\deg f_1 = \deg f_2 = 2^{t-1} - 1$. Az f_1 együtthatói, az a egész, majd pedig f_2 együtthatói egyszerű együttható összehasonlítással megkaphatók. Az a értékét tudjuk úgy választani, hogy f_2 főegyütthatója 1 legyen. Végezzük el ugyanezt a felbontást az f_1, f_2 polinomokra, majd a belőlük kapott újabb polinomokra, stb., amíg ez lehetséges. Végeredményben összesen t ilyen menetben kifejezzük $f(x)$-et $x^{2^j} + b$, (b egész) alakú polinomok szorzatainak összegeként. Egy menetben összesen n új együtthatót kell meghatározni.

Példa: Nézzük az $f(x) = x^7 + x^5 + 2x + 1$ polinom felbontását. Itt $n = 7$ és $t = 3$. Az első lépés után $f(x) = (x^4 - 1)(x^3 + x) + x^3 + 3x + 1$, azaz $f_1(x) = x^3 + x$,

és $f_2(x) = x^3 + 3x + 1$. E két polinomot tovább bontjuk ($t = 2$). Az eredmény: $f_1(x) = (x^2 + 1)x$ és $f_2(x) = (x^2 + 2)x + x + 1$. Végül

$$f(x) = (x^4 - 1)(x^2 + 1)x + (x^2 + 2)x + x + 1.$$

Az f-nek az így kapott felírását használva egy kiértékelés 5 szorzást jelent. Két szorzással kapjuk az x^2 és x^4 hatványokat, majd elvégezzük a felbontásban szereplő további három szorzást.

Vegyük szemügyre általánosabban is a felbontást használó kiértékelés költségét. Legyen $M(t)$ = az előbbi felbontás szerint a szorzások száma egy kiértékelésnél. Legyen továbbá $N(t) = M(t) - t + 1$. A kiértékelést a c helyen úgy fogjuk végezni, hogy először kiszámítjuk a $c^2, c^4, \ldots, c^{2^{t-1}}$ hatványokat (ez $t - 1$ szorzás), utána pedig a felbontásból adódó $c^{2^j} + b$ alakú tényezőket szorozzuk össze. Az ilyen szorzások száma lesz $N(t)$. Nyilvánvaló, hogy $N(1) = 0$. A felbontás definíciójából kiolvasható, hogy $N(t) \leq 2N(t-1) + 1$, ha $t > 1$, hiszen az $f_1(c)$ és $f_2(c)$ kiszámításához használt szorzásokon felül még egy további szorzás szükséges. A rekurzióból kézenfekvő indukcióval kapjuk, hogy $N(t) \leq 2^{t-1} - 1$. Innen az összes szorzások száma $M(t) \leq 2^{t-1} + t - 2 = \frac{n-3}{2} + \log_2(n+1)$. A k darab kiértékelés összesen $k(\frac{n-3}{2} + \log_2(n+1))$ szorzással elvégezhető.

Feladat: Mutassuk meg, hogy a fenti séma szerinti egyetlen kiértékelésnél legfeljebb $\frac{3n-1}{2}$ összeadás elegendő.

Megjegyzések:

1. V. Pan egyik tételéből következik hogy egy n-edfokú $f(x)$ polinom (f főegyütthatója tetszőleges) egyszeri kiértékeléséhez a legrosszabb esetben legalább n szorzás/osztás kell.
2. Ismeretes olyan prekondícionált algoritmus is, amellyel egy helyettesítés $\lfloor \frac{n}{2} \rfloor + 2$ szorzással kiszámolható. Másfelől bármilyen jól prekondícionáljuk is a polinomot, mindig lesz olyan helyettesítés, ami nem kapható meg kevesebb, mint $\lfloor \frac{n}{2} \rfloor$ szorzással (E. G. Belaga tétele).
3. Az egyszerűség kedvéért, hogy elég legyen csak a szorzásokra figyelnünk, nem foglalkoztunk az előkészítő munka költségével. A gazdaságos k értékek meghatározásánál erre is tekintettel kell lennünk.

3. Mintaillesztés szövegben

A feladat, amivel itt foglalkozunk, alapvető fontosságú a szövegeket tároló, kezelő rendszerek szemszögéből: egy adott jelsorozatot kell megkeresnünk egy hosszú jelsorozatban. Az ilyen és hasonló igények a közhelyszerű számítógépes alkalmazások mellett (pl. keresők, szerkesztők) fontos szerepet játszanak a genetikai kutatások egyik irányzatában, ahol a genetikai kódot roppant méretű, az A,G,C,U,T

jelekből formált szavakkal modellezik. A következőkben az egyik leghatékonyabb ötletet mutatjuk be, melynek lényeges része a keresett minta prekondícionálása. Először fogalmazzuk meg pontosan a feladatot:

Probléma: Adottak a Σ véges abc betűiből alkotott $s = s_1 \cdots s_n$ és $m = m_1 \cdots m_d$ szavak ($s_i, m_i \in \Sigma$). Találjuk meg m első részszóként való előfordulását az s szóban.

Úgy is fogalmazhatunk, hogy a legrövidebb olyan x szót keressük, melyre alkalmas y szóval teljesül, hogy $s = xmy$. Például az $s = balalajka$ szóban az $m = la$ minta a harmadik betűvel kezdődően fordul elő először részszóként, míg az $m = lak$ szó egyáltalán nem fordul elő.

Természetesen kínálkozik az a megoldás, hogy $i = 1, 2, \ldots, n - d + 1$-re sorban megvizsgáljuk, hogy az s-nek az i-edik betűjével kezdődő d hosszúságú részszava egyenlő-e m-mel. Ez az eljárás legrosszabb esetben $d(n - d + 1)$ betű-összehasonlítást igényel.

Első hallásra talán meglepő, hogy van a feladatra ennél jóval gyorsabb módszer is. A Knuth–Morris–Pratt-algoritmus, amit ismertetni fogunk, lineáris idejű, vagyis $O(n + d)$ összehasonlítást használ.

Definíció: *Az y szó az x szó* kezdőszelete (végszelete)*, ha van olyan nem üres z szó, hogy $yz = x$ (illetve $zy = x$).*

Definíció: *Az y szó az x szó* lábfeje*, ha y az x leghosszabb olyan kezdőszelete, ami egyben végszelete is. (Az y tehát a leghosszabb olyan w szó, melyre alkalmas nem üres z, z' szavakkal $x = wz = z'w$ teljesül.)*

Például a *bababa* szó lábfeje *baba*. Egy szónak önmaga nem lehet a lábfeje. A továbbiakban a leírás szempontjából kényelmes lesz a C programozási nyelvből ellesett egyik jelölés. A szavakat egészekkel indexelt, Σ típusú tömböknek is tekintjük. Így például az s szó második betűje hivatkozható mint $s[2]$, a harmadik helytől az ötödikig terjedő részszava pedig mint $s[3 : 5]$.

Mielőtt forró fejjel elkezdenénk az m szót az s különböző részeivel hasonlítgatni, először elemezzük, feltárjuk az m szerkezetét. Ezáltal bizonyos összehasonlítások eredményét többször is felhasználhatjuk a keresésben, számottevően redukálva az összköltséget. A prekondícionálás az m szó előfeldolgozását fogja jelenteni. Pontosabban fogalmazva kitöltjük a $P[1 : d]$ egész típusú tömböt, ahol

$$P[j] \stackrel{\text{def}}{=} \text{az } m[1:j] \text{ szó lábfejének a hossza.}$$

A P kitöltésére szolgáló módszer egyszersmind takaros példa a dinamikus programozás alkalmazására, amennyiben a P már ismert részét használjuk a még

ismeretlen értékek meghatározására. Nyilvánvalóan $P[1] = 0$. A módszer leírásakor kényelmes lesz még a $P[0] = 0$ megállapodás. Tegyük fel ezután, hogy $j > 1$ és a $P[1 : j-1]$ résztömböt már kitöltöttük. Ekkor $P[j]$ így számítható:

1. $i := j - 1$; $P[j] := 0$; $P[0] := 0$; 2. Ha $m[j] = m[P[i] + 1]$ akkor legyen $P[j] := P[i] + 1$, különben, ha $i > 0$, akkor legyen $i := P[i]$, és menjünk vissza a 2. lépésre.

Az eljárás megértéséhez jelölje w az $m[1 : j]$ szó lábfejét és l a w hosszát. Az l értéket kell meghatározni, ugyanis $P[j] = |w|$. Először megjegyezzük, hogy a $P[j]$ értéke nyilván legfeljebb $P[j-1]+1$ lehet, hiszen ha az $m[1 : l]$ szó azonos az $m[j-l+1 : j]$ szóval, akkor $m[1 : l-1] = m[j-l+1 : j-1]$ is igaz. Ennek a lehetőségét nézzük meg először a második sor összehasonlításával. Ha itt nemleges a válasz, akkor sem üres a kezünk. Tudjuk, hogy $P[j] \leq P[j-1]$, ami azt jelenti, hogy a $w[1 : l-1]$ szó kezdőszelete $m[1 : P[j-1]]$-nek és végszelete az $m[j - P[j-1] : j-1]$ szónak. De az utóbbi két szó a P tömb definíciója szerint azonos: $m[1 : P[j-1]] = m[j - P[j-1] : j-1]$. Ezzel az észrevétellel l meghatározását visszavezettük egy kisebb hasonló feladatra. A w ezután annak a szónak (is) a lábfeje, amit úgy kapunk, hogy az $m[1 : P[j-1]]$ szó végére fűzzük az $m[j]$ betűt. Ezt a visszavezetést valósítja meg az $i := P[i]$ értékadás.

Példa: Vegyük szemügyre a módszer működését az $m = ababaa$ bemenettel. Tegyük fel, hogy a $P[1 : 5]$ résztömböt már kitöltöttük, és éppen $P[6]$ meghatározásánál tartunk. Tudjuk tehát, hogy $P[1] = P[2] = 0$, $P[3] = 1$, $P[4] = 2$ és $P[5] = 3$. A keretes eljárás indításakor $j = 6$ és emiatt $i = 5$. A 2. sorban az $m[6]$ és $m[4]$ betűket hasonlítjuk össze. Mivel ezek nem egyeznek, az $i = 3$ értékkel térünk vissza a 2. sorba. A teszt eredménye újfent negatív: $m[6] \neq m[2]$. Ezáltal megint a *különben* ágra kerülünk, ahonnan az $i := 1$ értékadás után jutunk a 2. sorba. Az összehasonlítás végre valahára egyezést mutat ($m[1] = m[6] = a$), tehát megtörténik a $P[6] := P[1] + 1 = 1$ értékadás.

A $P[\]$ tömb kitöltésének időköltsége nyilvánvalóan arányos a szükséges betűösszehasonlítások számával, így csak az utóbbiakat számoljuk össze. Jelölje k_j a $P[j]$ számításához felhasznált betűösszehasonlítások számát. Ebbe $P[1 : j-1]$ költségét nem értjük bele. Először megmutatjuk, hogy $k_j \leq P[j-1] - P[j] + 2$. Világos, hogy a 2. sor tesztjét pont k_j-szer hajtjuk végre, hiszen a kódban csak itt van betűösszehasonlítás. Vizsgáljuk meg, hogy mi történik ezalatt a $P[i]$ értékkel. Ez kezdetben $P[j-1]$, majd minden további ilyen teszt előtt legalább eggyel csökken. A végső értéke, amit f-fel jelölünk, ezért nem lehet több, mint $P[j-1] - k_j + 1$. Tudjuk azt is, hogy $P[j]$ értéke vagy $f + 1$ lesz, vagy f aszerint, hogy a legutolsó betűösszehasonlítás sikeres volt-e, vagy sem. (Utóbbi esetben $f = 0$.) Ezeket

összevetve $P[j] \leq P[j-1] - k_j + 2$, amiből $k_j \leq P[j-1] - P[j] + 2$.

Az összehasonlítások száma mármost

$$\sum_{j=2}^{d} k_j \leq \sum_{j=2}^{d} (P[j-1] - P[j] + 2) = P[1] - P[d] + 2(d-1) \leq 2(d-1).$$

Az előfeldolgozás tehát a minta hosszával arányos költséggel, vagyis lineáris időben elvégezhető.

Ezután a Knuth–Morris–Pratt-algoritmus általános lépését mutatjuk be. Tegyük fel, hogy az m mintát éppen az $s[i]$ betűvel kezdődően próbáljuk illeszteni, és már tudjuk, hogy $s[i] = m[1], \ldots, s[i+j-1] = m[j]$, azaz a minta első j betűjénél illeszkedést találtunk.

1. Ha $j = d$, akkor készen vagyunk, találtunk egy m-mel azonos részszót s-ben. Ha $j < d$, akkor (feltéve, hogy $i + j \leq n$) összehasonlítjuk az $m[j+1]$ és $s[i+j]$ betűket. Ha ezek egyenlők, akkor $j := j + 1$, és a lépést befejeztük. Ha $m[j+1] \neq s[i+j]$, akkor két esetet különböztetünk meg:
2. (a) Ha $j = 0$ (azaz éppen m első betűjénél tartunk), akkor m-et egy hellyel jobbra csúsztatjuk: $i := i + 1$, és a lépést befejeztük.
(b) Ha $j > 0$, akkor az m mintát $j - P[j]$ hellyel jobbra csúsztatjuk: $i := i + j - P[j]$ és $j := P[j]$, és a lépést befejeztük.

A lényeges ötlet a 2.(b) lépésben lapul. A $P[j]$ érték mondja meg, hogy mi az a legkisebb esélyes mennyiség (nevezetesen $j - P[j]$), amivel érdemes eltolni jobbra az m mintát. Vegyük észre, hogy az eltolás után az m első $P[j]$ betűje egyezik az s megfelelő betűivel. A fontos ebből, hogy az s szóban nem kell visszalépnünk. Az $i + j$ mennyiség ugyanaz a 2. (b) lépés előtt és után.

Példa: Az $s = cdcbabababa$ szövegben keressük az $m = bababc$ mintát. Tegyük fel, hogy m-et már az s negyedik betűjétől kezdve próbáljuk illeszteni, és éppen túl vagyunk az első öt betű hasonlításán. Mindez számokkal azt jelenti, hogy $i = 4$ és $j = 5$:

			i							
c	d	c	b	a	b	a	b	a	b	a
			b	a	b	a	b	c		
							$i+j$			

A következő összehasonlításnál kiderül, hogy $s[9] \neq m[6]$, és ezzel a 2. (b) ágba kerülünk. Hasznát vesszük az eddigi sikeres összehasonlításoknak. Ezekből tudjuk, hogy $s[4:8] = m[1:5]$. A következő esélyes illeszkedés helyét a P

tömbből tudhatjuk meg. Mivel $P[5] = 3$, a legkisebb esélyes eltolás az eddigi ismeretek alapján 2. A pusztán eggyel való eltolással nem érdemes próbálkozni; ha ott volna illeszkedés, akkor $m[1:5]$ lábfejének a hossza 4 lenne. A 2. (b) lépés után ez lesz a helyzet:

$$\begin{array}{ccccccccccc}
 & & & & & i & & & & & \\
c & d & c & b & a & b & a & b & a & b & a \\
 & & & & & b & a & b & a & b & c \\
 & & & & & & & i+j & & &
\end{array}$$

A lépés befejeztével $i = 6$ és $j = 3$. Az s szövegben nem kellett visszalépnünk, a mintát pedig jobbra mozdítottuk.

A módszer időigényének nagyságrendi becsléséhez itt is elég a betűösszehasonlítások számát vizsgálni. Ezt nagyban megkönnyíti az a tény, hogy az eljárás során az i és az $i + j$ mennyiségek *sohasem csökkennek*. Kezdetben $i = 1$ és $j = 0$, tehát $i + j = 1$. Egy lépés (ami legfeljebb egy betűösszehasonlítást jelent) megtétele után vagy i vagy pedig $i + j$ értéke nő. Mivel $i \leq n$ és $i + j \leq n$, a lépések és így a betűösszehasonlítások száma legfeljebb $2(n-1)$. A módszer tehát összegészében – a prekondícionálás idejét is beleértve – lineáris uniform költségű.

10.

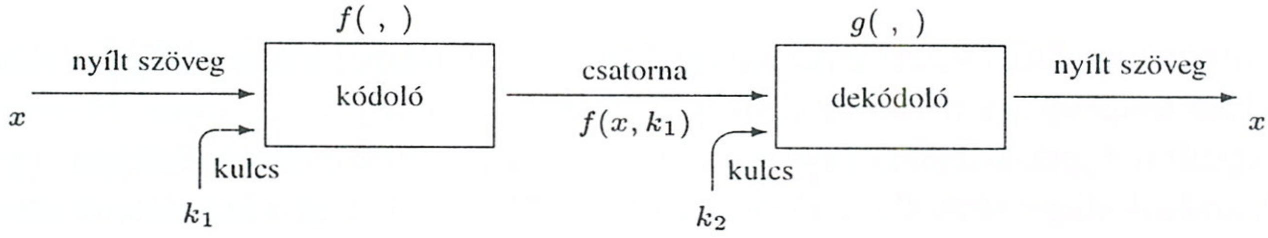
Nyilvános kulcsú titkosírások

A meglepetés kulcsa a gyorsaság ötvözése titoktartással.
CARL VON CLAUSEWITZ

Itt rövid ízelítőt adunk a kriptográfia, a titkos információcsere elemeiből. Az illetéktelen hozzáféréssel szemben biztonságos kommunikáció sokáig elsősorban a politikusok és a katonák ügye volt. Ma, az elektronikus hálózatok világméretű térhódításával alapvető, mindennapi igénnyé vált a biztonságos és gyors információcsere. Ennek megfelelően a kriptográfia komoly tudományággá nőtt; se szeri, se száma a protokolloknak és algoritmusoknak. Utóbbiak egy része az algoritmusok bonyolultságának elméletén alapul. A kriptográfusok sajátos, fordított szemlélettel nézik a bonyolultsági eredményeket, amennyiben elsősorban a hatékonyan nem kezelhető problémák érdekesek számukra. Ami a gyors algoritmusok tervezésén fáradozók számára átok, az számukra áldás lehet. Több olyan kriptográfiai módszer ismeretes, amelynek a biztonságossága, „feltörhetetlensége" egy algoritmikus probléma nehézségén alapul. Bizonyos mértékig ilyen módszer a gyakorlatban is igen népszerű RSA-algoritmus, amit vázolni fogunk, érdekes alkalmazását adva ezzel néhány eddig megismert fogalomnak és eredménynek.

10.1. Kriptográfia - a titkosírások tudománya

Először szólunk néhány szót a kriptográfiáról általában. A titkos információcsere következő egyszerű és általános modelljét fogjuk használni.



Az x a kódolatlan – szokásos szakkifejezéssel: *nyílt* – üzenet, amit el szeretnénk juttatni a *vevőhöz*. Köztünk és a vevő között képzeljük el a *csatornát*, amin az üzenetnek át kell jutnia. A csatorna a kommunikációs rendszernek az a része, ahol az üzenethez illetéktelenek hozzáférhetnek. Ez ellen úgy védekezünk, hogy a csatornán az x üzenet *kódolt* változatát küldjük át, amit a másik oldalon a vevő megfejt, visszakapva az eredeti nyílt üzenetet. A kódolást, a titkosítást egy f eljárással (függvénnyel) végezzük. Az f egy k_1 *kulcs* segítségével végzi az átalakítást. Az f függvénynek a kódolandó szöveg és a k_1 információ a paraméterei. Az eredményül kapott rejtjelezett $f(x, k_1)$ üzenet megy át a csatornán. A vevő oldalán a titkos üzenet megfejtését, a *dekódolást* szintén egy kulccsal (jelölje ezt k_2) vezérelt g eljárás végzi. A g az y rejtjelezett üzenetből visszaadja a $g(y, k_2)$ nyílt szöveget. Az egymásnak megfelelő f, g eljárásokkal és k_1, k_2 kulcspárral szemben nyilvánvaló követelmény, hogy $g(f(x, k_1), k_2) = x$ teljesüljön minden x üzenetre. Jóval kevésbé formalizálható az a követelmény, hogy a rendszer biztonságos legyen, vagyis pusztán a csatornán átmenő kódolt $f(x, k_1)$ üzenetből ne lehessen kitalálni az x nyílt üzenetet.

A Vernam-kódoló

A teljes x nyílt üzenet és a $k_1 = k_2 = d$ kulcsok egyforma hosszú bitsorozatok, és $f(x, d) := x \oplus d$. Itt $\oplus$ a bitenkénti kizáró vagy (XOR) műveletet jelenti, d pedig egy „véletlen" bitsorozat. Nyilvánvaló, hogy $g = f$ használható dekódoló függvényként, hiszen egyfelől $d \oplus d$ sorozat csupa nullából áll, másfelől az $\oplus$ művelet asszociatív. A Vernam-kódoló egy régi, sokat használt eljárás. Hátránya, hogy a meglehetősen terjedelmes d kulcsot a kommunikáció megkezdése előtt valahogy (pl. futárral) biztonságosan el kell juttatni a másik félhez.

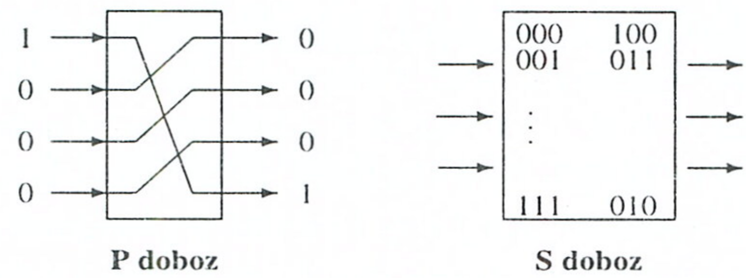
A továbbiakban olyan rendszerekkel foglalkozunk, melyekben az f kódoló függvény az üzenet rögzített (mondjuk k) hosszúságú darabjait kódolja el, és a titkos üzenet ezen elkódolt darabok egymásutánja. Hasonlóképpen működik a g dekódoló.

A DES rendszer

Röviden vázoljuk az IBM által kifejlesztett, alkalmas paraméterekkel az USA-ban

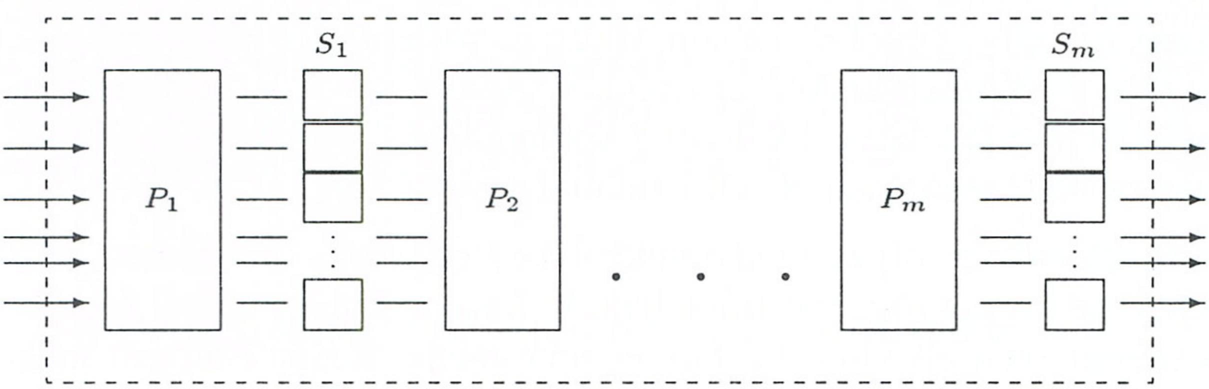
szabványosított DES (Data Encryption Standard) rendszer elemeit. Egyik kódoló-dekódoló alapeszköz a *P-doboz*. Egy P-doboz az $x_1 x_2 \cdots x_k$ input bitsorozatot egy rögzített σ permutációval az $x_{\sigma(1)} x_{\sigma(2)} \cdots x_{\sigma(k)}$ bitsorozattá alakítja.

A másik alapeszköz az *S-doboz*. Egy S-doboz kulccsal vezérelhető. A kulcs egy beállítása mellett a doboz egy $s : \{0,1\}^k \to \{0,1\}^k$ kölcsönösen egyértelmű (azaz invertálható) leképezést valósít meg. A következő ábra egy 4 bites ($k = 4$) P-dobozt és egy 3 bites ($k = 3$) S-dobozt mutat.



A P-doboz itt ciklikusan balra lépteti az input biteket; az S-doboznál pedig a benne lakozó s függvényt igyekeztünk szemléltetni. Tehát például $s(000) = 100$, $s(001) = 011$.

A DES ajánlás 64 bites nyílt üzenetek 64 bites rejtjelezett üzenetté való kódolását javasolja. E célra 64 bites P-dobozokat és 4 vagy 8 bites S-dobozok 16 illetve 8 magasságú tornyait használja. Ilyen kódoló egységeket kell sorban egymás után fűzni. Jelölje $P_1, \ldots P_m$ a használt P-dobozokat, $S_1, \ldots S_m$ az S-dobozok tornyait. A 64 bites x üzenetből a szintén 64 bites $S_m(P_m(\ldots S_1(P_1(x) \ldots)$ rejtjelezett üzenetet kapjuk.



Az IBM Lucifer elnevezésű titkosítási rendszere hasonló elrendezésben 128 bites P-dobozokat és 4 bites S-dobozok (32 magasságú) tornyait használja.

10.2. Nyilvános kulcsú kriptográfia

A régi vágású kriptográfiai rendszerek komoly gondja a kulcsok kezelése. A Vernam-kódoló jól mutatja ezt. A rendszer használata előtt az adónak és vevőnek meg kell egyeznie a használt d sorozatokat illetően. Az ehhez szükséges információcserének is biztonságosnak kellene lennie, hiszen ha valaki megneszeli a d sorozatokat, akkor könnyen meg tudja fejteni az $x \oplus d$ alakú üzeneteket. A nyilvános kulcsú rendszerek elegáns megoldást adnak a kulcsok kezelésére. Működésükhöz, mint látni fogjuk, nincs szükség előzetes titkos információcserére.

Legyenek a kommunikációs hálózat résztvevői $A_1, \ldots, A_n$. A_i -nek egy titkos D_i és egy nyilvános E_i kulcsa van. Az E_i kulcs mindenki számára ismert, mondjuk benne van a telefonkönyvben. Szintén nyilvános (és minden résztvevő számára ugyanaz) az f kódoló függvény, melyre minden i index és x input szó (üzenet) esetén teljesül, hogy

$$f(f(x, E_i), D_i) = f(f(x, D_i), E_i) = x.$$

E kikötések azt fogalmazzák meg, hogy f használható dekódoló függvényként is; ezenfelül (E_i, D_i) és (D_i, E_i) összetartozó kulcspárok abban az értelemben, hogy ha az egyik kulcsot használjuk kódolásra, akkor a másik használható dekódolásra. Teljesülnie kell még két fontos további követelménynek:

$\mathbf{K}_1$: Adott x, E_i bemenetre $f(x, E_i)$ hatékonyan számítható; adott x, D_i bemenetre $f(x, D_i)$ hatékonyan számítható.
$\mathbf{K}_2$: Pusztán x és E_i ismeretében $f(x, D_i)$ nem számítható hatékonyan; ugyanígy, ha csak x és D_i ismert, akkor $f(x, E_i)$ nem számítható ki hatékonyan.

A $\mathbf{K}_1$ feltétel azt jelenti, hogy a kódolás, illetve dekódolás gyorsan végezhető. Ez a feltétel viszonylag egyszerűen teljesíthető. Jóval több gond van a $\mathbf{K}_2$ feltétellel, ami azt célozza, hogy a rendszer biztonságos, nehezen feltörhető titkos kommunikációt tegyen lehetővé. A $\mathbf{K}_2$ tulajdonságot az ismert rendszerek úgy igyekeznek elérni, hogy adott x, E_i esetén $f(x, D_i)$ meghatározása egy nehéz (pl. ismereteink szerint polinom időben nem kezelhető) algoritmikus probléma megoldását jelentse. Ugyanez vonatkozik természetesen az x, D_i párokra és az $f(x, E_i)$ értékekre is.

Tegyük fel, hogy van egy ilyen f függvényünk, A_i rendelkezik a D_i titkos kulccsal, és a telefonkönyvben megtalálhatók az E_i nyilvános kulcsok. Nézzük, miként oldható meg két kommunikációs alapfeladat ebben a keretben.

1. *A_1 titkos üzenetet küld A_2-nek.* A_1 az x nyílt szövegből az A_2 nyilvános kulcsával az $y = f(x, E_2)$ rejtjelezett üzenetet képezi, és ezt küldi el a csatornán.

A_2 ezt a saját titkos kulcsával dekódolhatja: $x = f(y, D_2)$. A $\mathbf{K}_2$ feltétel biztosítja, hogy az A_2-n kívül más ezt az y üzenetet nem tudja megfejteni.

2. *A_1 aláírt titkos levelet küld A_2-nek.* Ekkor A_1 az $y = f(f(x, D_1), E_2)$ üzenetet küldi el. Más szóval A_1 először kódolja x-et a D_1 titkos kulcsával, majd ennek az eredményét az A_2 nyilvános kulcsával. A fogadó oldalon A_2 így veheti le a két lakatot: $x = f(f(y, D_2), E_1)$. Az első lépésben a birtokában levő D_2 kulccsal visszakapja az $f(x, D_1)$ szót, ami aztán az E_1 nyilvános kulccsal kezelhető. Az aláírást az jelenti, hogy a $\mathbf{K}_2$ feltevés szerint a második lépésben csak az E_1 kulcs fogja nyílt szöveggé alakítani $f(y, D_2)$-t.

10.3. A Rivest–Shamir–Adleman- (RSA-) kód

Az előbb körvonalazott nyilvános kulcsú kriptográfiai séma egy nevezetes megvalósítását szeretnénk bemutatni . Mielőtt nekivágnánk, kis kitérőt teszünk. Szükségünk lesz az *euklideszi algoritmus* egyik változatára, amivel egészek legnagyobb közös osztóját lehet gyorsan meghatározni.

Legyenek $a_0 \geq a_1$ legfeljebb k bit hosszúságú természetes számok. Képezzük az a_i egészeket maradékos osztással a következők szerint:

$$\begin{aligned} a_0 &= q_1 a_1 + a_2, \\ a_1 &= q_2 a_2 + a_3, \\ &\vdots \\ a_{n-2} &= q_{n-1} a_{n-1} + a_n, \\ a_{n-1} &= q_n a_n + a_{n+1}, \end{aligned}$$

úgy, hogy $|a_{i+1}| \leq |a_i/2|$ teljesüljön minden $i > 0$ indexre. A sorozatot addig képezzük, amíg nullát nem kapunk; vagyis $a_{n+1} = 0$. Ez szükségképpen bekövetkezik, mégpedig gyorsan. Ha a két kezdő számunk legfeljebb k bit hosszúságú, akkor $n \leq k$, hiszen minden lépésben – kivéve esetleg az utolsót – legalább egy bittel rövidebb egészet kapunk. Az a_{i+1} az a_i és a_{i-1} számokból maradékos osztással kapható; a q_i vagy $\lfloor a_{i-1}/a_i \rfloor$ lesz, vagy pedig $\lceil a_{i-1}/a_i \rceil$. Ezek közül valamelyikkel biztosan teljesül az $|a_{i+1}| \leq |a_i/2|$ egyenlőtlenség.

A sorozat végétől az eleje felé lépdelve látható, hogy $|a_n|$ osztója mindegyik a_i-nek, így a_1-nek és a_0-nak is. Ha pedig a b pozitív egész osztója a_0-nak és a_1-nek is, akkor a sorozat elejétől indulva kapjuk, hogy b osztója mindegyik a_i-nek, ezért $|a_n|$-nek is. Vagyis $|a_n|$ éppen a_0 és a_1 legnagyobb közös osztója. A legnagyobb közös osztó tehát polinom időben kiszámítható.

Feladat: Mutassuk meg, hogy polinom időben kaphatunk olyan c_0 és c_1 egészeket, melyekre $c_0 a_0 + c_1 a_1 = lnko(a_0, a_1)$, továbbá $|c_0| \leq a_1$ és $|c_1| \leq a_0$. Ezek

alapján elmondhatjuk, hogy a_0 és a_1 legnagyobb közös osztója hatékonyan kifejezhető a két szám egész együtthatós lineáris kombinációjaként, mégpedig nem túl nagy együtthatókkal. (Az euklideszi sorozat elejéről indulva sorra fejezzük ki az a_i számokat a_0 és a_1 segítségével.)

Megjegyezzük még, hogy a legkisebb nemnegatív maradékokat használó euklideszi sorozat (az $|a_{i+1}| \leq |a_i/2|$ követelmény helyett $0 \leq a_{i+1} < a_i$ szerepel) is elég gyorsan eléri a nullát; ennek az igazolása azonban valamivel bonyolultabb.

Ezután ismertetjük a Rivest–Shamir–Adleman- (széles körben használt rövidítéssel RSA-) kriptorendszert. Ahogy korábban megállapodtunk, legyenek $A_1, \ldots, A_n$ a kommunikációs hálózat résztvevői. Először A_i generál két nagy prímszámot, p_i-t és q_i-t (mondjuk 500 bináris jegyűeket). Ez a 9.4.3. pontban látottak szerint hatékonyan megtehető véletlen választásokkal és prímteszteléssel. Legyen $m_i = p_i q_i$. Ezen felül A_i választ egy véletlen e_i természetes számot, melyre $1 < e_i < m_i$ és $lnko(e_i, \phi(m_i)) = 1$. Itt ϕ az Euler-függvényt jelenti. Tudjuk, hogy $\phi(m_i) = (p_i - 1)(q_i - 1)$. Végül kiszámít egy olyan $0 < d_i < m_i$ egészet, melyre $e_i d_i \equiv 1 \pmod{(p_i - 1)(q_i - 1)}$. Az e_i ellenőrzése, majd pedig d_i számítása az euklideszi algoritmussal történhet. Az utóbbihoz olyan c_0 és c_1 egészeket keresünk, melyekre $c_0 e_i + c_1 \phi(m_i) = 1$, $|c_0| \leq \phi(m_i)$ és $|c_1| \leq e_i$ (lásd az euklideszi algoritmussal kapcsolatos feladatot). A $d_i := c_0 \pmod{\phi(m_i)}$ választás nyilvánvalóan jó lesz.

Az A_i titkos kulcsa a $D_i = (d_i, m_i)$, nyilvános kulcsa pedig az $E_i = (e_i, m_i)$ pár. Az E_i kulcsot közzéteszi, a D_i-t, pontosabban d_i-t pedig megtartja magának.

Feltesszük, hogy a nyílt üzenet olyan x egészek sorozata, melyekre $0 \leq x < m_i$ és $lnko(x, m_i) = 1$ teljesül. Az első feltétel pusztán azt jelenti, hogy az üzenetet blokkonként akarjuk kezelni. A második feltétel nem támaszt komoly megszorítást; m_i-hez képest elenyésző azoknak a $[0, m_i]$ intervallumba eső x egészeknek a száma, amelyek oszthatók p_i-vel vagy q_i-vel.

Az f kódoló/dekódoló függvénynek három argumentuma van. Ezekből az utolsó kettő együtt alkotja a kulcsot. Az f a nyílt szöveg egy x blokkját alakítja át rejtjelezett blokkä:

$$f(x, e_i, m_i) := x^{e_i} \pmod{m_i}.$$

Így néz ki az x-nek az A_i nyilvános kulcsával kódolt változata. Feltesszük, hogy az eredmény a legkisebb nemnegatív maradék: $0 \leq f(x, e_i, m_i) < m_i$. Ahogy korábban utaltunk rá, ugyanezzel a függvénnyel végezzük a dekódolást is. A_i az általa kapott titkos üzenet y blokkját ($0 \leq y < m_i$) az alábbi módon fejtheti meg:

$$f(y, d_i, m_i) := y^{d_i} \pmod{m_i}.$$

Nézzük ezután az előző pontban megfogalmazott követelményeket! Tudjuk, hogy $e_i d_i = 1 + \phi(m_i) q$ teljesül alkalmas q egészre. Ezt, és a számelméletből jól

ismert Euler–Fermat-tételt használva

$$f(f(x, e_i, m_i), d_i, m_i) = (x^{e_i})^{d_i} \equiv x^{e_i d_i} \equiv x \cdot (x^{\phi(m_i)})^q \equiv x \cdot 1^q \equiv x \ (\mathrm{mod}\ m_i),$$

vagyis a rejtjelezés után a dekódolás tényleg visszaadja az eredeti üzenetet.

A szorzás kommutativitásából közvetlenül adódik az E_i, D_i kulcsok felcserélhetőségét jelentő

$$f(f(x, e_i, m_i), d_i, m_i) = f(f(x, d_i, m_i), e_i, m_i) = x$$

azonosság. Ami a $\mathbb{K}_1$ feltételt illeti, f hatékonyan számítható gyors hatványozással. A $\mathbb{K}_2$ feltételt az „biztosítja", hogy az e-edik gyökvonásra mod m_i nem ismert polinom idejű módszer (randomizált sem), ha m_i összetett, és nem tudjuk a prímosztóit.

Feladat: Mutassuk meg, hogy p_i, q_i és e_i ismeretében d_i hatékonyan meghatározható. Tehát az m_i felbontását ismerők A_i titkos kulcsát könnyen kideríthetik.

Megjegyzés: Az RSA biztonságosságát illetően elmondhatjuk, hogy a rendszer eddig ellenállt a feltörési kísérleteknek, és szép, sikeres gyakorlati alkalmazásai vannak. Ugyanakkor *nincs bizonyítva*, hogy a feltörése algoritmikusan nehéz feladat lenne. Nem tudjuk egyfelől, hogy az egészek felbontása (m_i ismeretében p_i és q_i meghatározása) nehéz-e. Másfelől az sincs bizonyítva, hogy az RSA elleni sikeres támadáshoz tényleg kell tudni egészeket felbontani; pusztán arról van szó, hogy más esélyes támadási irányról nincs tudomásunk.

Vannak olyan, az RSA-nál bonyolultabb rendszerek, amelyek feltörése bizonyítottan legalább olyan nehéz, mint az egészek prímtényezős felbontásának a feladata.

Példa: Szeretnénk itt szemléltetni az RSA-rendszer működését. Az olvasó kényelméért eltekintünk a többszáz bites prímek használatától; beérjük öt bittel is. További egyszerűsítésként elhagyjuk az i indexet (A_i helyett A-t, p_i helyett p-t írunk, stb.).

Legyen $p = 11$ és $q = 17$. Ekkor az m modulus értéke $11 \cdot 17 = 187$. A $\phi(m)$ értéke $10 \cdot 16 = 160$. Legyen az e nyilvános kulcs 9, és határozzuk meg a d titkos kulcsot. E célból az euklideszi algoritmust hívjuk segítségül. Az $a_0 = 160$, $a_1 = 9$ bemenettel a számítás így alakul:

$160 = 18 \cdot 9 - 2,$

$9 = (-4) \cdot (-2) + 1,$

$-2 = (-2) \cdot 1 + 0.$

Innen látjuk egyrészt, hogy $lnko(\phi(m), e) = lnko(160, 9) = 1$ tényleg igaz. Másrészt az első egyenlőségből -2-t kifejezve és a másodikba helyettesítve $9 = (-4) \cdot$

$(160 - 18 \cdot 9) + 1$ adódik, amiből átrendezéssel kapjuk, hogy $-71 \cdot 9 + 4 \cdot 160 = 1$. A titkos kulcs tehát $d = 89 \equiv -71 \pmod{160}$.

Egy az A-nak szánt x üzenetblokk kódja az $f(x, 9, 187) := x^9 \pmod{187}$ szó lesz. Például az $x = 3$ üzenetnek $3^9 = 3^4 \cdot 3^4 \cdot 3 = 81 \cdot 81 \cdot 3 \equiv 16 \cdot 3 \equiv 48 \pmod{187}$ lesz a kódja. Az y rejtjelezett blokk dekódolása, visszafejtése az $f(y, 89, 187) := y^{89} \pmod{187}$ függvénnyel történik.

Tárgymutató

Ki nevek a végén
PARTI NAGY LAJOS

JELÖLÉSEK

2^{I^*} *211*
$\prec$ *269*
$\subset$ *253*
$A[i:j]$ *11*
$\alpha(m)$ *165*
$bal(x), jobb(x)$ *58*
$C_M(x)$ *230*
C_N *91*
C'_N *91*
$\mathcal{R}$ *202*
$\mathcal{RE}$ *202*
χ_L *215*
$\chi(g)$ *16*
co$TIME(t(n))$ *252*
coX *211*
$C(x)$ *231*
$elem(x)$ *58*
f_M *194*
F_n *56*
$FSPACE(s(n))$ *249*
$FTIME(t(n))$ *249*
$G=(V,E)$ *15, 112*
γ *44*
H_f *219*
$h_i(K)$ *90*
$h(K)$ *87*
I^* *193*
L_d *208*
L_h *216*
L_M *194*
L_u *209*
M_w *206*
$m(f)$ *71*
$\mid H \mid$ *112*
$\mid y \mid$ *230*
$m(x)$ *71*
$NTIME(t(n))$ *257*
$O(\cdot)$ *11*
$o(\cdot)$ *11*
ϕ *56*
$\mathbb{R}$ *11*
$\mathbb{R}^+$ *11*
Σ^* *26*
$SPACE(s(n))$ *249*
$\subseteq$ *17*
$s(x)$ *58, 78*
$TIME(t(n))$ *248*
ü (üresjel) *192*
$\mathbb{Z}$ *11*
$\mathbb{Z}^+$ *11*

0-9

2-3-fa *64, 69, 70*
2-SAT $\rightarrow$ *kielégíthetőség*

3-DH → *3 dimenziós házasítás*
3 dimenziós házasítás (3-DH) **282**, *285*, *295*
3 ház – 3 kút → *gráf*
3-SAT → *kielégíthetőség*
3-SZÍN → *színezhetőség*
80-20 szabály → *eloszlás*

A

abc-sorrend *26*, *83*
adatszerkezet *37*
adattömörítés *102*
adjacencia-mátrix *16*, **113**, *116*, *123*, *125*, *157*, *262*
algoritmus *9*, *13*, *18*, *191*, **203**, *247*
 hindu–arab- *21*
 rendezési → *rendezés*
alsó becslés *19*, *33*, *37*, *44*, *47*
alsó korlát (hálózatban) *186*
általánosított Riemann-sejtés *320*
alternáló
 erdő → *erdő*
 út → *út*
álvéletlen *315*
 próbálás *93*
apa (fában) *38*
apró paraméter *304*
aranymetszés *56*
artikulációs pont *145*
asszociatív szabály *76*
átlagos költség *27*, *32*, *43*, *63*, *86*, *89*, *91*, *94*, *98*, *99*
Avizienis algoritmusa *23*
AVL-fa **71**, *77*, *79*

B

Batcher módszere → *összefésülés*
befejezési szám **130**, *133*, *138*, *143*, *144*
beillesztés → *beszúrás*
bejárás → *fa*, *gráf*
bejáró algoritmus → *gráf*, *fa bejárása*
Bellman–Ford-módszer **120**, *123*, *139*, *302*
belső
 csúcs *38*, *64*, *65*, *69*
 memória *51*, *70*, *90*, *95*, *110*
 pont módszer *294*
beszúrás *31*, *37*, **40**, *42*, **57**, **66**, *70*, *73*, *80*, *83*, *85*, *86*, *91*, *98*, *159*
 naiv *61*, *63*, *73*
beszúrásos rendezés → *rendezés*
B-fa *69*
bináris
 fa → *fa*
 keresés → *keresés*
 keresőfa → *keresőfa*
binomiális együtthatók *302*
binsort → *rendezés (láda-)*
blokkoló folyam → *folyam*
Boole
 -formula *272*
 kielégíthető → *kielégíthetőség*
 -mátrix *125*
Borel–Cantelli-lemma *238*
Borůvka módszere **155**, *166*
branch-and-bound → *elágazás és korlátozás*
breadth-first-search (BFS) → *gráf bejárása*
buborék-rendezés → *rendezés*

C

Cantor-féle átlós módszer **204**, *208*, *253*
Carmichael-szám → *pszeudoprím*
Church tétele *217*

Church–Turing-tézis *203, 229, 239, 242*
CNF → *konjunktív normálformájú*
coNP **257**, *261, 263–265, 267, 270, 271, 316*
Cook–Levin-tétel *272*
coRP *315*

CS

csomósodás *95*
 elsődleges *93*
 másodlagos *93*
csoport *265*
csúcs *15*, **111**
 kritikus *141*
 súlya *58, 78*
csúcsösszevonás *67*
csúcsvágás *67*

D

DAG → *gráf*
dekódolás *102, 108, 329*
depth-first-search (DFS) → *gráf bejárása*
DES (Data Encryption Standard) *329*
determináns *311, 313*
d-fa → *fa*
diagonális nyelv **208**, *209*
Dijkstra módszere **116**, *119, 122, 124, 139, 140, 150, 157, 160, 298*
dinamikus
 halmaz *98*
 Huffman-kód → *Huffman-kód*
 programozás *123, 299*, **302**, *324*
Dinic módszere *179*, **181**, *184, 189, 254, 265*
diofantikus
 egyenlet *218, 228*
 halmaz *219*
d-kupac → *kupac*
Dominóprobléma *222*
dupla forgatás → *forgatás*

E

Edmonds–Karp-algoritmus **179**, *180*
egyenletes eloszlás → *eloszlás*
ekvivalencia
 -osztály *112, 142*
 -reláció *112, 142*
él **111**
 kritikus *141, 176, 180*
 színezése → *piros-kék módszer*
 telített → *telített pontpár*
elágazás és korlátozás *299*
eldönthetetlen *191*, **215**, *216, 219, 222, 226, 227, 234*
eldönthető **203**
élek osztályozása **131**, *138*, **144**, *148*
 előreél **132**, *148*
 faél **131**, **145**, *148, 150*
 keresztél **132**, *148, 150*
 visszaél **132**, *136, 138, 145, 148*
elérési gyakoriság → *keresési gyakoriság*
élidegen utak *183, 184*
Éllefogás feladat **282**, *286*
éllefogó halmaz *185, 282*
ellenség-módszer *21, 28*
ellipszoid-módszer *294*
éllista **114**, *119, 129, 143, 150, 158, 161, 170, 254*
 fordított *139*

eloszlás *29*
 80-20 szabály szerinti *101*
 egyenletes *100, 312*
 Zipf *100*
előreél → *élek osztályozása*
elsődleges csomósodás → *csomósodás*
elsőfokú csúcs *152*
élsúly *110*, **112**, *115, 135, 139, 151, 160, 166, 167*
 egységnyi *128, 149, 289*
 kis egész *165*
 negatív *120, 122*
 nemnegatív *115, 120, 124, 139, 140*
 pozitív egész *289*
elvágó pont → *artikulációs pont*
erdő **112**, *131, 154*
 alternáló *169*
 feszítő → *feszítő erdő*
erőforrás-kiosztás *167*
erősen összefüggő
 gráf *141, 145*
 komponens *141*
euklideszi algoritmus *332*
Euler
 -állandó **44**, *100*
 -bejárás *308*
 –Fermat-tétel *334*
 -függvény *333*
explicit halmaz *221*
exponenciális idő *252, 271, 301*
EXPTIME **252**, *271*

F

fa *38, 58*, **112**, *120, 148, 151, 163, 256, 277*
 2-3- → *2-3-fa*
 AVL- → *AVL-fa*
 B- → *B-fa*
 bejárása **59**, *111, 127*
 inorder *59, 79*
 Lindström- *309*
 mélységi *131*
 postorder *59, 131, 321*
 preorder *59, 131, 321*
 bináris **38**, **58**, *78, 103*
 teljes *38, 59*
 feszítő- → *feszítőfa*
 kiegyensúlyozott *71, 77, 79*
 magassága (szintszáma) *59, 61, 63–68, 70–72, 77, 78*
 S- → *S-fa*
 teljes d-fa *41*
faél → *élek osztályozása*
felszivárog → *kupac*
Fermat
 -tanú *317*
 -teszt *316*
 tétele *94, 316*
feszítő erdő
 mélységi *131, 145*
 szélességi *148*
feszítőfa *111, 146*
 mélységi *132, 145*
 minimális költségű **151**, *156, 160, 254*
 kulcsmanipulációs módszerrel *166*
 más költségfogalommal *166*
 összehasonlítással *166*
 szélességi *149*
Fibonacci
 -hash-elés → *hash-elés*
 -szám **55**, *72*
FIFO-lista → *sor adatszerkezet*
first fit *306*
 decreasing *307*
fiú (fában) *38*

Floyd módszere **122**, *125–127*, *302*, *304*
FOGYASZT → *kupac*
fok **112**
folyam *111*, *150*, **171**, **172**, *188*, *264*
 blokkoló *181*
 értéke *172*
 maximális **173**, *175*, *179*, *180*, *182*, *184–186*, *254*, *265*
 nulla- *173*, *176*, *179*, *181*, *186*
 vágáson áthaladó *174*, *185*
Ford–Fulkerson
 -algoritmus **178**, *179*
 -tétel *174*, **177**, *184*, *185*, *264*
forgatás *73*, *76*, *81*, *82*
 dupla *73*
forrás *115*, **171**, **172**
FP *246*, **249**, *254*, *267*, *268*, *290*, *295*
futam *52*
független
 élhalmaz *305*
 pontrendszer *280*, *282*
 maximális (MAXFTLEN) **280**, *282*, *300*
függvény kiszámítása *213*

G

Gauss-elimináció *311*
gráf *14*, *15*, *110*, **111**, *254*, *255*, *262*, *263*, *278*, *280*, *285*
 3 ház – 3 kút *263*
 ábrázolása *113*
 bejárása *111*
 mélységi *111*, **127**, **128**, *136*, *138*, *143*, **144**, *145*, *147*
 szélességi *111*, *128*, **146**, *149*, *170*, *180*
 centruma *126*
 irányítatlan *18*, *110*, **111**, *120*, *144*, *145*, *148*, *151*, *262*, *269*, *286*, *289*
 irányított *18*, *110*, **111**, *115*, *135*, *141*, *144*, *147*, *172*, *183*, *188*, *263*, *269*, *286*
 végtelen *224*
 irányított körmentes (DAG) **135**, *140*, *181*, *288*, *296*
 topologikus rendezése *136*, **137**, *141*, *144*, *152*, *296*
 páros **16**, **167**, *169*, *184*, *282*, *313*
 ritka **115**, *119*, *158*
 súlyozott → *élsúly*
 sűrű *119*, *160*
 teljes 5-ös *263*

GY

gyors hatványozás *265*, *317*, *334*
gyorsítási tétel *198*
gyorsrendezés → *rendezés*
gyökér **38**, *58*, *59*, *64*, *67–69*, *81*, *84*, *103*, *146*, *148*, *163*, *224*

H

hálózat **172**
 alsó korlátokkal *186*, *188*
hálózati folyam → *folyam*
Hamilton-kör *140*, *255*, **262**, *264*, *269*, **285**, *289*, *308*
 irányított *263*, *269*, *285*
 minimális költségű → *Utazó ügynök*
harmonikus szám **44**, *100*
hash-elés *86*, *98*
 Fibonacci- *97*
 kettős **95**, *96*, *97*
 nyitott címzés *87*, **90**, *95*
 vödrös *87*, **88**, *95*

hash-függvény *87, 90, 95, 97*
megfelelő *87*, **95**
hash-kódolás → *hash-elés*
Hátizsák feladat **289**, *303*
apró értékkorláttal *304*
apró súlykorláttal *304*
HB-fa *77*
Hilbert 10. problémája *191, 217*
hindu–arab-algoritmus → *algoritmus*
hóeke modell *54*
Horner-elrendezés *322*
Huffman
-fa *103*
-kód *102*
dinamikus *105*
hurokél *113, 132, 133*

I

időigény **197**, *247*
index-szint *70*
inorder → *fa bejárása*
fonal *61*
input (bemenet) *13*
hossza → *szó hossza*
szalag *193*
interpolációs keresés → *keresés*
intervallumlistázás (TÓLIG) *57, 61, 68, 70, 98*
invariancia-tétel *230, 316*
inverz Ackermann-függvény *165*
irányítatlan gráf → *gráf*
irányított
gráf → *gráf*
kör → *kör*
körmentes → *gráf*
út → *út*
írógép-paradoxon *229*
irracionális szám *97*
irreflexív reláció *25*
iteráció *37*
izolált csúcs *103*

J

javító gráf *175, 180*
javító út → *út*
Johnson módszere *159*
jól karakterizált nyelv *263, 316*

K

kanonikus felsorolás **204**, *214, 219, 233*
kapacitás **171**, **172**
egész *178, 254, 264*
irracionális *179*
kritikus *176*
vágásé → *vágás*
karakterisztikus függvény *215*
Karp-redukció **268**, *272, 276–278, 281–283, 285, 286, 289–292, 295*
kék → *piros-kék módszer*
kékes él → *piros-kék módszer*
kék fa → *piros-kék módszer*
keresés *27*, **57**, *61, 65, 70, 80, 85, 86, 89, 98*
k-adik elemé *44*
bináris **28**, *32, 267, 290, 297*
interpolációs *29*
lineáris **27**, *31*
sikeres/sikertelen *91, 94, 95, 99*
szekvenciális *87, 89, 99*
keresési
feladat *266*
gyakoriság *80, 100*
keresőfa **57**, *83, 86, 98, 297*
átlagos költsége *63*
bináris *60, 63, 71, 77, 78, 80*
-tulajdonság *60*

keresztél → *élek osztályozása*
kétrészes gráf → *gráf (páros)*
kettős hash-elés → *hash-elés*
kezdőszelet *84, 103, 324*
kiegyensúlyozott
 ábrázolás *22*
 fa → *fa*
kielégíthetőség (SAT) **272**, *295*
 k-SAT *277*
 3-SAT **276**, *277, 278, 283*
 2-SAT *277*
kifejezésfa *76, 277*
KIFORDÍT művelet → *S-fa*
kifordított fa → *S-fa*
kínai maradéktétel *318*
Kirchoff törvénye *171, 172*
kis Fermat-tétel → *Fermat tétele*
kiszámíthatatlan *205, 208, 233*
kiszámítható *190, 191*, **202**, **203**, *208, 210*
klasztereződés → *csomósodás*
klikk → *teljes részgráf*
Knuth–Morris–Pratt-algoritmus *324*
kódolás *102, 106, 329*
kódolt (rejtjelezett) üzenet *329*
Kolmogorov
 -bonyolultság *191*, **229**, *231, 234, 236, 316*
 -véletlen sorozat *237*
kombinatorikus optimalizálás *111, 171*
komplementer
 gráf *282*
 nyelv **211**, *252, 263–265*
 nyelvosztály **211**, *252*
komponens → *összefüggő komponens*
kongruens szám *227*
konjunktív normálformájú
 Boole-formula *276*
 k-CNF *276*
 3-CNF *277*
 2-CNF *277*
konzervatív rendezés → *rendezés (stabil)*
korlátozó heurisztikák *300*
König
 -akadály *17, 186, 264*
 -lemma *224, 226*
 minimax tétele **184**, *264, 282*
kör **112**, *153, 161, 162, 262*
 egyszerű *112, 151, 152*
 irányítatlan *132*
 irányított *135, 263*
 negatív összsúlyú *115, 121, 122*
 legrövidebb *150*
körmentes *112, 132, 135, 151, 162*
közelítő módszer *299*, **305**
közvetlen elérésű gép (RAM) *191*, **239**, *253, 255*
kriptográfia *328*
kritikus
 kapacitás → *kapacitás*
kromatikus szám *16*
Kruskal módszere *155*, **160**, *162, 164–166, 254, 298, 308*
kulcs *26*, **29**, *37, 46, 57, 58, 64, 69, 84, 86, 88, 90, 95, 329*
 nyilvános *331, 333*
 összetett *48*
 titkos *331, 333*
kulcsmanipuláció
 élsúlyokkal → *feszítőfa*
 rendezésnél → *rendezés*

kupac **37**, *41, 45, 54, 58, 59, 100, 111, 119, 158, 159, 161, 254*
d-kupac *41, 119, 159, 160*
BESZÚR eljárás → *beszúrás*
-építés *39, 42*
felszivárog eljárás *39, 42*
FOGYASZT eljárás **41**, *42, 119, 160*
MINTÖR eljárás → *minimumtörlés*
-tulajdonság *38, 41*
kupacos rendezés → *rendezés*
Kuratowski tétele *263*
különleges út → *út*
külső tár *51, 52*
kvadratikus idő *254*
kvadratikus maradék próba *93, 96*

L

lábfej *324*
Ládapakolás *291, 295, 306*
ládarendezés → *rendezés*
Lagrange tétele *220*
láncolás → *hash-elés (vödrös)*
lapelérés *52, 69, 70, 89*
lap (külső táron) *52, 69, 88*
lapösszevonás *70*
lapvágás *70*
Las Vegas **315**, *320*
-teszt *315*
lefogó élek *184*
lefogó ponthalmaz → *éllefogó halmaz*
leghosszabb út → *út*
legkisebb elem → *minimumkeresés*
legnagyobb elem → *maximumkeresés*
legnagyobb közös osztó (lnko) *317, 332*
legrövidebb út → *út*
Lempel–Ziv–Welch-módszer *88*, **106**
leszármazott (fában) *132, 321*
levél *38, 59, 64, 65, 69, 84, 103*
lexikografikus rendezés → *rendezés*
Lindström-bejárás → *fa bejárása*
lineáris
idő *21, 40, 45, 47, 49, 60, 119, 129, 139, 143, 149, 160, 165, 166, 248, 253, 254*
keresés → *keresés*
próbálás **91**, *95, 96*
programozás *292*
egész értékű *294*
lista adatszerkezet *37*
literál **272**, *276, 279, 283*
logaritmikus
költség **241**, *253, 255, 262*
tár *249*
logikai cím *88*
Lucifer rendszer *330*

M

magyar módszer **168**, *184, 254*
maradékos osztás *96*
maradékosztály *94, 265*
másodlagos csomósodás → *csomósodás*
Matijaszevics tétele *220*
mátrix *113*
MAXFTLEN → *független pontrendszer*
maximális
folyam → *folyam*
független → *független*
párosítás → *párosítás*
teljes részgráf → *teljes részgráf*

maximumkeresés *44, 57, 61, 70, 98, 127*
MAXKLIKK → *teljes részgráf*
Megállási probléma *191*, **216**, *217, 220, 234*
mélységi
 bejárás (keresés) → *gráf bejárása*
 szám **130**, *133, 144, 146*
Menger tétele **184**, *264*
minimális
 késésszámú ütemezés → *ütemezés*
 költségű feszítőfa → *feszítőfa*
 vágás → *vágás*
minimax tétel *173*
 Ford–Fulkerson- **174**, *175*, **177**, *184, 185, 264*
 König- **184**, *264, 282*
 Menger- **184**, *264*
minimumkeresés *44, 57, 61, 70, 79, 98, 118, 119, 127, 159, 161*
minimumtörlés *37*, **40**, *41, 42, 45, 54, 119, 159, 161*
mintaillesztés *323*
modell *12, 14*
módosítás *57, 86, 89*
moduláris
 hatványozás *266*
 összeadás *90*
mohó elemzés *107*
mohó módszer *111, 116, 128, 156, 160, 298*
multiplikatív inverz *94*
munkaszalag *197*
mutató (pointer) *58, 59, 61, 64, 69, 82, 84, 88, 110, 114, 138, 160, 163, 165, 239, 243, 255, 321*

N

nemdeterminisztikus
 felismerés *258*
 Turing-gép → *Turing-gép*
növelés → *Dinic módszere*
növelő út → *út*
NP *246*, **257**, *258, 261, 262, 265, 267, 269, 271, 272, 295, 314, 316*
NP-nehéz **276**, *298*
NP-teljes *246*, **271**, *272, 275, 277, 278, 281–283, 285, 286, 289–292, 295, 300, 301, 303, 308*
NTG → *Turing-gép (nemdeterminisztikus)*
nulla-folyam → *folyam*

NY

nyelő **171**, **172**
nyelv *193*
 felismerése **194**, *211, 213, 267*
 -osztály *211*
nyílt üzenet *329*
nyilvános kulcsú titkosírás *331*
nyitott címzés → *hash-elés*
nyomulás → *Dinic módszere*

O

on-line algoritmus *166*
optimalitás elve *111*, **122**, *123, 302*
oszd meg és uralkodj *28, 35, 42, 297*
osztómódszer *96*
output (eredmény) *13*
 szalag *193*

Ö

önszervező *80, 101, 164*
ős (fában) *146, 148, 321*

összefésülés *35, 38*
páros-páratlan (Batcher) *50*
többfázisú (polifázisú) *55*
összefésüléses rendezés → *rendezés*
összefüggő
gráf **112**, *132, 141, 145, 146, 151, 152, 154, 160, 166*
komponens **112**, *132, 142, 145, 148, 166*
összefűzés (rendezett listáké) *79*
összehasonlítás *25, 33*
összenyomhatatlan **232**, *237*
összenyomható *191, 229*
összetettségi teszt *316, 319*
összetett szám *265, 316, 317, 319*

P

P *246*, **248**, *254, 257, 261, 264–267, 270, 271, 277, 290, 314, 320*
palindróma (tükörszó) *235, 253*
parciálisan rekurzív függvény **203**, *213*
parciális függvény *202*
párhuzamos algoritmus *22*
gráf bejárására *128*
min. feszítőfára *155*
összeadásra *21*
rendezésre/összefésülésre *26, 50*
páros gráf → *gráf*
párosítás *111, 150*, **167**
maximális **167**, *184, 254, 282, 306*
párosított csúcs *167*
teljes **17**, *313*
páros-páratlan összefésülés → *összefésülés*
partíció *162, 167, 174*
Partíció feladat *291, 292*
Pascal-háromszög *302*
patt *136*
PCP → *Post megfeleltetési problémája*
P-doboz *330*
permutáció *33, 90, 93*
PERT
-gráf *135*, **140**
-módszer **140**, *176*
pillanatnyi helyzet *250, 256, 273*
piros → *piros-kék módszer*
piros-kék módszer **153**, *156, 166*
kék
él *153, 156, 160*
erdő *166*
fa **154**, *156, 160–162*
szabály **153**, *156*
kékes él *157, 159*
piros
él *153, 160*
szabály *153*
színezetlen él *153*
takaros színezés *153*
pitagoraszi számhármas *218*
pointer → *mutató*
polifázisú összefésülés → *összefésülés*
polinom *218, 311*
-azonosság tesztelése *311*
foka *218*
idő **248**, *249, 254, 255, 265, 282, 294, 298, 301, 304, 306, 313–316, 320, 321, 331, 332, 334*
kiértékelése *322*
tár *252*
polinomiálisan összehasonlítható *245*

Pontos fedés hármasokkal (X3C) 282, **285**, 290
Post megfeleltetési problémája 226
postorder → *fa bejárása*
Pratt tétele 265, 267
prefix → *kezdőszelet*
kód 103
prekondícionálás 299, **321**
preorder → *fa bejárása*
primitív gyök 265
Prim módszere 155, **156**, 159, 160, 166, 254, 298, 308
prímszám 93, 96, 221, 265, 316, 333
keresése 320
-tétel 320
prímtényezős felbontás 266, 290
prímteszt 316, 320
prioritás 37
prioritási sor 38
próbasorozat 91, 93–95
PSPACE **252**, 261, 271
pszeudoprím (Carmichael-szám) 318

R

Rabin–Miller
-tanú 320
-teszt **318**, 320
radix rendezés → *rendezés*
RAGASZT művelet → *S-fa*
RAM → *közvetlen elérésű gép*
randomizált → *véletlent használó*
redukált gráf 143
rekurzió 37
rekurzív
függvény **203**, 213
halmaz 219
nyelv **202**, 208, 210, 211, 213, 253
rekurzíve felsorolható
halmaz 219
nyelv **202**, 208, 210, 211, 213, 214
relatív prímek 95, 97
rendezés 25, 98, 99
Batcher- → *összefésülés*
beszúrásos **31**, 89, 99
bináris kereséssel 32
lineáris kereséssel 31
buborék- **30**, 33, 44
gyors- **42**, 64, 298, 310
kulcsmanipulációs 26, **46**
kupacos 38, **41**
külső tárakon 51
láda- **47**, 48, 83
lexikografikus **26**, 48, 83, 214, 219, 254
összefésüléses **35**, 46, 50, 51, 297
külső tárakon 52
összehasonlítás alapú 26, **29**, 33, 35
radix **48**, 83, 165
stabil (konzervatív) 31
rendezési reláció 25
rendezett
halmaz 25
lista 79, 161
típus **25**, 46, 47, 57
részbenrendezés 296
részfa 58, 70, 71, 77, 132, 145
-lemma **134**, 137, 138, 144
részgráf 151
részhalmaz 162
Részhalmazösszeg probléma (RH) 290, 291, 295
ritka gráf → *gráf*
robusztusság 253, 254, 261

RP *314, 319, 320*
 -teszt *315*
RSA-rendszer *328,* **332**

S

SAT → *kielégíthetőség*
Schwartz-lemma *312*
S-doboz *330*
S-fa *80, 101, 164*
síkba rajzolhatóság **263**, *282*
síkgráf → *síkba rajzolhatóság*
sor adatszerkezet *138, 147, 149*
splay tree → *S-fa*
stabil rendezés → *rendezés*
standard ábrázolás *22*
Stirling-formula **32**
súgásszalag *258*
súlyfüggvény → *élsúly*
súlyra kiegyensúlyozott fa (SK-fa) *77*
sűrű gráf → *gráf*

SZ

számítási út *256, 259*
számtani sorozat *97*
szekvenciális
 elérés *52*
 keresés → *keresés*
szélességi
 bejárás (keresés) → *gráf bejárása*
 feszítő erdő → *feszítő erdő*
 számozás *148*
szimmetrikus differencia *168*
szimplex-módszer *293*
szimuláció *198, 242, 253, 259*
színezés *16, 262*
 2 színnel *254, 280*
 3 színnel (3-SZÍN) *255,* **262**, *264,* **278**, *281, 282, 295, 301*
 éleké → *piros-kék módszer*
színezetlen él → *piros-kék módszer*
szintszám → *fa magassága*
szó *83, 193*
 hossza *85, 195, 197, 204, 214, 229, 230, 232, 234, 235, 241, 247, 253, 255, 256, 259, 260, 262, 265, 266, 268, 269, 273, 304, 314*
szófa **83**, *106*
szomszéd *67, 116, 119, 129, 147*
szomszédossági mátrix → *adjacencia-mátrix*
szorzómódszer *96*
szótárszerű rendezés → *rendezés (lexikografikus)*
szuperforrás *18*
születésnap-paradoxon *87*

T

T. Sós Vera tétele *97*
takarítás → *Dinic módszere*
tanú **259**, *260–267, 270, 272, 273, 277, 281, 283, 285, 290, 295, 311, 315, 317*
tanú-tétel *259*
tár–idő-tétel *216, 250*
tárigény **197**, *247*
távolság (gráfban) **115**, *122, 149, 180*
telített pontpár (él) *172, 180*
telítettségi tényező *91*
teljes
 fa → *fa*
 ötszög → *gráf*
 párosítás → *párosítás*
 rendezési reláció *25*

részgráf *281*
maximális (MAXKLIKK) *255*, **281**, *282*, *296*
Termékszerkezet-feladat *292*
testvér (fában) *67*
TÓLIG → *intervallumlistázás*
topologikus rendezés → *gráf*
többfázisú összefésülés → *összefésülés*
törlés *57*, **66**, **67**, *70*, *76*, *80*, *86*, *89*, *91*, *92*, *98*
naiv *61*, *73*
TÖRÖLT jel *92*
törzstényezős felbontás *334*
tranzakció *135*
tranzitív
lezárt *125*
reláció *25*
Turing-gép *190*, **191**, *247*, *255*, *316*
egyszalagos *199*, *235*, *253*
felismert nyelve *194*
időkorlátos *247*
kétszalagos *200*, *253*, *258*
kiszámított függvény *194*
kódja (leírása) *205*, *206*, *208*
nemdeterminisztikus (NTG) *255*
számolási ideje *197*, *247*
tárigénye *197*
tárkorlátos *248*
univerzális **205**, *208*, *209*, *216*, *229*
Turing tétele *210*, *217*
Tutte tétele *313*
tükörszó → *palindróma*

U

unáris ábrázolás *304*
uniform költség *241*, *280*, *288*
UNIÓ-HOLVAN adatszerkezet *111*, **162**
univerzális
kereső feladatok *271*
nyelv **209**, *210*
Turing-gép → *Turing-gép*
univerzum *37*, *46*, *86*
út *80*, *84*, *103*, **112**, *115*
alternáló *168*
egyszerű **112**, *121*, *139*, *151*
hossza *115*, *180*
irányított **112**, *122*, *134*, *141*, *149*, *176*
végtelen *224*
javító *168*
kritikus *141*
különleges *117*
leghosszabb *289*
DAG-ban *139*, *288*
legrövidebb *111*, **115**, *122*, *128*, *149*, *304*
DAG-ban *139*
nyomonkövetése *120*, *122*, *125*, *140*
növelő *176*, *179*, *180*
legrövidebb *179*
Utazó ügynök probléma *140*, *255*, **285**, *289*
euklideszi *308*
útösszenyomás *164*

Ü

ütemezés *135*, *188*
minimális késésszámú *296*
ütközés **87**, *96*
feloldása *87*, **90**

V

vágás **174**, *185*
kapacitása **174**, *185*

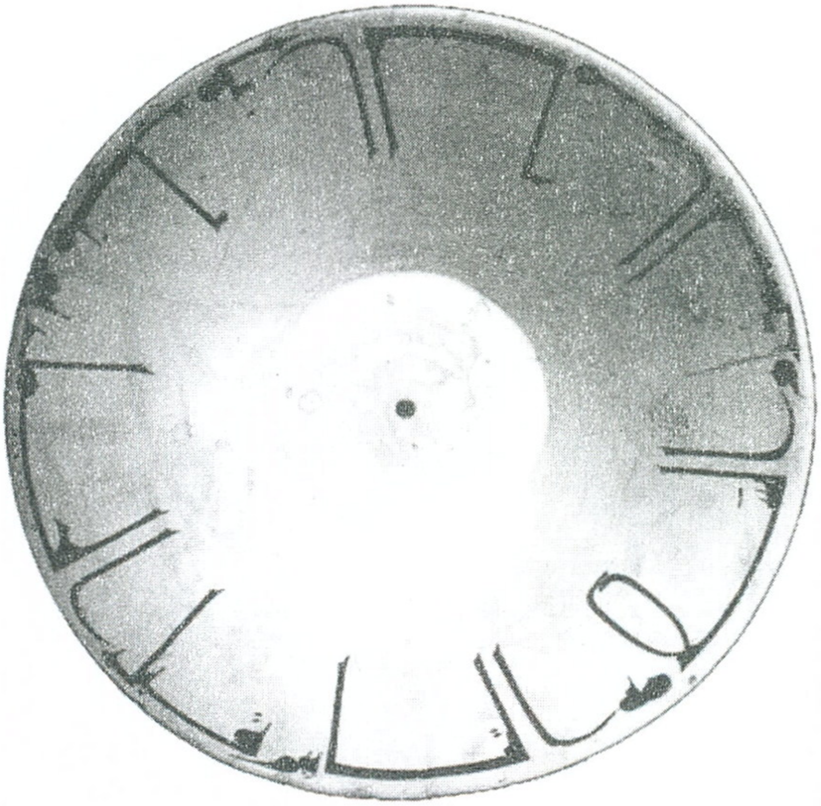
minimális **175**
valós idejű *98*
várakozási gráf *135*
végszelet *324*
végtelen
ciklus *125*, **192**, *194*, *209*, *213*, *216*, *250*, *257*
út → *út*
véletlen sorozat → *Kolmogorov*
véletlent használó módszer *42*, *166*, *298*, **310**, **314**
Vernam-kódoló *329*
visszaél → *élek osztályozása*
von Mises-paradoxon → *születésnap-paradoxon*
vödör *88*
-katalógus **88**, *95*, *99*
vödrös hash-elés → *hash-elés*

W

Warshall algoritmusa *126*

Z

zár *136*
Zipf eloszlás → *eloszlás*



Hétszer mérj, egyszer vágj!

TYPOTEX

1024 Budapest, Retek u. 33–35. Tel./fax: 316-3759
e-mail: typotex@euroweb.hu, www.vision.euroweb.hu/typotex

Felsőfokú matematika könyveink

Stoyan G. – Takó G.: **Numerikus módszerek I–III.**
Kovács Margit: **A nemlineáris programozás elmélete**
Stoyan Gisbert: **Numerikus és statisztikai módszerek, grafika**
Laczkovich Miklós: **Sejtés és bizonyítás**
A. Borovkov: **Matematikai statisztika**
Lovász László: **Kombinatorikai problémák és feladatok**

A könyvek megvásárolhatók a kiadóban (1024 Budapest, Retek u. 33–35.) és a Műszaki Könyváruházban. (1061 Budapest, Liszt F. tér 9.)

Kiadja a TypoTEX Kft. Elektronikus Kiadó
Felelős kiadó: Votisky Zsuzsa
Szedés és tördelés: Ivanyos Gábor
A borítót Debre Ferenc tervezte
Ívterjedelme: 24,6 A/5 ív

Készült a Múzsák Kiadó Nyomdájában
Felelős vezető: Várhelyi István

Algoritmusokkal és adatszerkezetekkel kapcsolatos ismeretekre, készségekre mindenkinek szüksége van, aki komolyan foglalkozik programozással és programok tervezésével. Ennek megfelelően kialakult egy eléggé letisztult törzsanyag, amit világszerte oktatnak a számítástechnikai, informatikai képzést nyújtó egyetemi szakokon. A könyv elsődleges célja ennek az anyagnak a feldolgozása.

A fontosabb témák a következők: rendezés, keresés, információtömörítés, gráfalgoritmusok, a kiszámíthatóság alapfogalmai, nevezetes bonyolultsági osztályok (P, NP) és algoritmustervezési módszerek. A bemutatott algoritmusok tárgyalását példák és feladatok teszik teljessé.

A könyv szerzői évek óta tanítanak algoritmikus témájú egyetemi tárgyakat a Budapesti Műszaki Egyetemen és az Eötvös Loránd Tudomány Egyetemen.

ISBN 963 9132 16 0

9 789639 132160

1600,-